Programmation impérative par raffinements avec l'assistant de preuve Coq

Boubacar Demba Sall

EDITE de Paris (ED130)

École doctorale Informatique, Télécommunications et Électronique

Programmation impérative par raffinements avec l'assistant de preuve Coq

THÈSE DE DOCTORAT EN INFORMATIQUE

Présentée par **Boubacar Demba SALL**

Soutenue publiquement le 01/10/2020 devant le jury composé de :

Mme Maria-Virginia APONTE *Maître de conférences, Conservatoire National des Arts et Métiers*	Examinatrice
M. Sylvain BOULMÉ *Maître de conférences, Université Grenoble Alpes*	Examinateur
M. Emmanuel CHAILLOUX *Professeur, Sorbonne Université*	Directeur de thèse
M. Tristan CROLARD *Professeur, Conservatoire National des Arts et Métiers*	Rapporteur
M. Marc FRAPPIER *Professeur, Université de Sherbrooke*	Rapporteur
M. Emmanuel LEDINOT *Directeur, System Software Engineering Laboratory* *Thales Research & Technology*	Examinateur
M. Antoine MINÉ *Professeur, Sorbonne Université*	Président
M. Frédéric PESCHANSKI *Maître de conférences, Sorbonne Université*	Co-Encadrant

À mes parents qui ont tant sacrifié pour leur famille.

Remerciements

En premier lieu, je voudrais vivement remercier tous les membres du jury. En particulier, je remercie M. Tristan Crolard et M. Marc Frappier d'avoir accepté la charge de rapporter ma thèse. Leur inspection minutieuse, ainsi que leurs critiques et suggestions constructives ont assurément permis d'améliorer ce manuscrit.

Je voudrais aussi exprimer toute ma gratitude et ma reconnaissance à Emmanuel et Frédéric, mes encadrants. Avant tout, je les remercie pour leur bienveillance, et leur soutien sans faille durant toute la thèse. Je les remercie également pour tous nos nombreux échanges scientifiques que j'ai beaucoup appréciés, et qui m'ont permis de construire et mener à bien un projet de recherche.

J'adresse mes très sincères remerciements à Mme Maria-Virginia Aponte, M. Sylvain Boulmé, M. Pierre Courtieu, M. Pierre-Évariste Dagand, M. Pascal Manoury, M. David Monniaux et Mme Marie-Laure Potet. Je vous suis infiniment reconnaissant d'avoir pris de votre temps pour m'écouter et partager avec moi votre avis éclairé.

Pour finir, je tiens à remercier tous les membres de l'équipe APR du Laboratoire d'Informatique de Paris 6 pour leur accueil très chaleureux. Grâce à vous, et en particulier à tous mes collègues du bureau 303, cette thèse a été une très belle aventure humaine et scientifique.

Résumé

Cette thèse s'intéresse à la programmation certifiée correcte dans le cadre formel fourni par l'assistant de preuve Coq, et conduite par étapes de raffinements avec l'objectif d'aboutir à un résultat correct par construction. Après avoir introduit le contexte général de la vérification des programmes nous posons le cadre technique dans lequel nos travaux ont eu lieu. Notamment, nous rappelons les fondements de la formalisation du processus de programmation par raffinements aussi bien du point de vue des théories basées sur le calcul des relations, que du point de vue des théories prédicatives telles que la logique de Hoare. Ces préliminaires sont suivies de la description du langage de programmation considéré qui est un langage impératif simple, avec affectations, sélections, séquences, et boucles. À ce langage, nous associons une sémantique relationnelle exprimée dans un cadre prédicatif plus adapté à un plongement dans la théorie des types, plutôt que dans le calcul des relations. Partant de cette sémantique, nous établissons formellement les propriétés générales de la notion de raffinement qui justifient l'approche de programmation par raffinements successifs. Par la suite, nous étudions la relation entre d'une part la sémantique prédicative et relationnelle que nous avons choisie, et d'autre part une approche plus classique dans le style de la logique de Hoare. En particulier, nous montrons que les deux approches ont en théorie la même puissance puisque tout raffinement valide dans une approche l'est aussi dans l'autre. La démarche que nous étudions consiste à certifier, avec l'aide d'un assistant de preuve, les raffinements successifs permettant de passer de la spécification au programme. Cependant, les démonstrations directes à partir des définitions de la notion de raffinement se révèlent très fastidieuses. Nous nous intéressons donc aussi aux techniques de preuve permettant en pratique d'établir la validité des raffinements. Globalement, il s'agit de simplifier les formules logiques qui émanent des énoncés de raffinement, et dont nous devons établir la validité. La simplification des formules logiques mentionnées passe par la formulation d'un calcul de la plus faible pré-spécification jouant ici le rôle du calcul de la plus faible précondition dans les approches plus classiques. Aussi, nous explorons une alternative à l'utilisation des notions d'invariant et de variant pour spécifier le comportement abstrait des boucles. Cette alternative est inspirée du point de vue relationnel sur les boucles, et elle s'appuie sur une spécification relationnelle des corps de boucle. Nous généralisons cette approche relationnelle de la correction des boucles, et nous montrons que notre généralisation est aussi bien correcte, qu'elle est complète relativement à l'expressivité du langage de spécification. Afin que l'articulation des étapes de raffinements reste aussi proche que possible de l'activité de programmation, nous formalisons un langage de développement qui permet de décrire l'arborescence des étapes de raffinements, ainsi qu'une logique permettant de raisonner sur les développements par raffinements. Grâce à cette logique, il est possible de garantir que les développements sont corrects par construction, et nous montrons que les propriétés de correction et de complétude que nous avons établies pour les méthodes de preuve des raffinements individuels, s'étendent à notre logique de raisonnement sur les développements globaux. Pour finir, nous présentons la mécanisation de nos travaux en mettant l'accent sur la mise à profit de l'infrastructure de l'assistant de preuve Coq pour faciliter l'encodage des programmes et la gestion des obligations de preuve correspondant aux différentes étapes de raffinement.

Abstract

This thesis investigates certified programming by stepwise refinement in the framework of the Coq proof assistant. This allows the construction of programs that are correct by construction. The programming language that is considered is a simple imperative language with assignment, selection, sequence, and iteration. The semantics of this language is formalized in a relational and predicative setting, and is shown to be equivalent to an axiomatic semantics in the style of a Hoare logic. The stepwise refinement approach to programming requires that refinement steps from the specification to the program be proved correct. For so doing, we use a calculus of weakest pre-specifications which is a generalisation of the calculus of weakest preconditions. Finally, to capture the whole refinement history of a program development, we formalize a design language and a logic for reasoning about program designs in order to establish that all refinement steps are indeed correct. The approach developed during this thesis is entirely mechanised using the Coq proof assistant.

Table des Matières

Chapitre 1

Introduction

Alors que l'un des premiers ordinateurs programmables modernes, le *Colossus*, fut construit en 1943 [18], dès 1949 Maurice V. Wilkes (Pionnier de l'informatique et prix Turing 1967) prenait conscience avec stupéfaction que la majeure partie de son temps de programmeur allait devoir être consacrée à la recherche et à la correction des erreurs qu'ils ne pourrait manquer de disséminer dans ses propres programmes [79]. Et pour cause, s'assurer qu'un programme est exempt d'erreurs est une tâche non triviale. En effet, si une exploration exhaustive de l'espace des états d'un programme est en théorie possible dans le cas d'un espace des états fini, en pratique une inspection individuelle de chaque état atteignable ne peut être effectuée en temps raisonnable du fait que l'espace à explorer est souvent gigantesque. Même pour des programmes de taille relativement modeste (quelques centaines de variables), seule une infime portion des états possibles pourrait être examinée si nous mobilisions toutes les ressources de calcul de la planète pendant un siècle. De plus, dans le cas des gros programmes, le fait qu'on puisse appréhender très clairement l'effet de chaque instruction considérée individuellement est insuffisant pour saisir le sens du programme dans sa globalité, ou lorsqu'il s'agit de vérifier que les programmes ont les propriétés escomptées. La raison en est qu'une partie de ce sens à tendance à se diluer lorsque le programmeur passe du *pourquoi*, du *but*, ou de l'*intention* qui représentent le sens, au *comment* qui se traduit par les instructions de programme. De manière générale, une vérification intelligente des programmes passe donc par un recouvrement du sens qu'il y a derrière les instructions de programme, c'est-à-dire par une démarche d'abstraction qui permet d'ignorer une masse de détails de fonctionnement, pour ne retenir que les aspects les plus pertinents vis-à-vis des propriétés souhaitées. Le contexte général de la vérification des programmes est donc marqué par les nécessités suivantes. D'une part, nous devons nous assurer que les abstractions que nous utilisons pour raisonner sur les programmes sont des abstractions correctes. Et, d'autre part, nous devons aussi nous assurer que les

raisonnements en question sont corrects. Cette thèse s'intéresse à l'approche de la programmation impérative par raffinements dans laquelle la pratique de la programmation est fortement dirigée par les nécessités de la vérification. Avant de présenter nos contributions, nous reviendrons d'abord plus en détail sur le contexte général de la vérification des programmes. Et pour finir nous exposerons le plan de cette thèse.

1.1 Contexte général

Bien que incomplète le test est la méthode de vérification la plus répandue. Dans le cas d'une vérification intelligente par tests, on peut dire qu'il y a une démarche d'abstraction sous-jacente. Dans un premier temps, cette démarche consiste à quotienter l'espace des états par la relation d'équivalence suivante : l'état x est équivalent à l'état y si et seulement si l'assertion "le programme exécuté sur x satisfait la spécification S" équivaut à l'assertion "le programme exécuté sur y satisfait S". Dans un second temps, faute de pouvoir effectuer tous les tests possibles, on se contente de tester un représentant de chaque classe d'équivalence. C'est ainsi qu'après avoir effectué un test sur une entrée donnée avec succès, on ne testera pas les autres entrées possibles considérées comme équivalentes. Cependant, encore faut-il que notre abstraction soit correcte, c'est-à-dire que toutes les classes d'équivalence aient effectivement au moins un représentant parmi les tests qui ont été sélectionnés. Si tel n'est pas le cas certaines erreurs ne seront probablement pas découvertes. De manière générale, nous ne disposons pas de solution efficace permettant de décider si un ensemble donné de tests suffit pour garantir la correction d'un programme. Dans le cas de certains systèmes paramétrés, on peut néanmoins noter quelques résultats intéressants comme par exemple dans le domaine des protocoles d'exclusion mutuelle [1], ou encore dans le cas de la modélisation des systèmes [55] avec le langage Alloy [38]. Ces résultats permettent de garantir, parfois automatiquement, qu'un système de taille n est correct en examinant seulement un modèle réduit du système de taille k, pour un k donné (beaucoup) plus petit que n.

Grâce aux méthodes de vérification automatique, comme par exemple la vérification de modèles [16] (ou *model checking*) ou encore l'analyse statique par interprétation abstraite [21, 22], des programmes de taille industrielle peuvent souvent être vérifiés sans qu'il soit nécessaire de communiquer formellement aux outils utilisés le but visé par les programmes. Pour être efficaces, ces méthodes utilisent des abstractions prédéfinies afin de contourner le problème de l'explosion de l'espace des états. Un avantage de ces méthodes est donc que l'effort d'abstraction demandé à l'utilisateur est minimal. Un autre avantage réside dans leur capacité à produire des contre-exemples en cas de détection d'erreur, ce

qui est très utile à la correction des erreurs. En conséquence de la nature relativement figée des abstractions utilisées, l'utilisateur est assez limité en expressivité quand aux propriétés qui peuvent être garanties par la vérification. De plus, les conclusions de la vérification peuvent aboutir à un nombre plus ou moins important de faux positifs en fonction de la précision des abstractions utilisées. Néanmoins des classes d'erreurs très fréquentes en programmation (débordement de tableaux, lecture de pointeurs invalides, etc.) peuvent être détectées automatiquement grâce aux outils qui permettent de mettre en œuvre ces méthodes.

Les approches déductives sont celles qui posent le moins de contraintes sur les propriétés des programmes auxquelles on peut s'intéresser puisque généralement les spécifications peuvent être écrites en logique du premier ordre. Ces approches par la preuve, ont été initiées par Turing [77], et sont ensuite devenues de plus en plus formelles grâce aux travaux de Floyd [29], Hoare [35] et Dijkstra [25]. L'idée visionnaire de Turing a consisté à associer des assertions aux points de programme afin de donner une représentation abstraite de l'ensemble des états accessibles aux dits points de programme. Cette démarche peut être comprise comme une manifestation de la nécessité de recouvrer le sens indispensable aux raisonnements sur les programmes en vue de leur preuve de correction. Le cas des invariants de boucles est sans doute l'exemple le plus patent de cette nécessité. Grâce à la représentation logique des états accessibles en un point de programme donné sous forme d'assertion, un problème de vérification peut être réduit en un problème de validation d'une formule logique. Cette validation peut ensuite être effectuée grâce à des outils automatiques tels que les solveurs SMT (Satisfiabilité Modulo Théories [10]), ou dans les cas les plus complexes, manuellement avec l'aide d'un assistant de preuve tel que Coq [76] ou Isabelle/HOL [62]. Les méthodes déductives permettent d'obtenir un haut niveau de fiabilité, mais l'effort de spécification et de preuve qu'elles nécessitent est jugé trop élevé sauf dans le cas de certains systèmes critiques.

De même le point de vue selon lequel la programmation devrait procéder par raffinements successifs, peut être compris comme une méthode permettant de conserver le sens derrière les instructions de programme, plutôt que de devoir plus difficilement recouvrer ce sens après-coup. Ce point de vue initié par Wirth [80] et Dijkstra [24], a ensuite été développé par Back [7], Morgan [57], Abrial [2] et beaucoup d'autres. La démarche de programmation par raffinements consiste à articuler de petites étapes dites de raffinements qui permettent de passer progressivement d'une spécification abstraite au départ, à une implémentation finale, en passant par autant d'étapes intermédiaires que nécessaires pour rendre explicite le processus de raisonnement qui a été suivi. Par la même occasion, ceci permet de mieux maîtriser la complexité en aidant à décomposer le problème

à résoudre. On peut considérer que chaque étape de raffinement comprend trois phases : spécification, implémentation et preuve. En particulier, les spécifications intermédiaires servent à décrire les objectifs visés par les différentes parties qui s'articulent pour former le programme. Idéalement, chaque étape de raffinement est formellement prouvée correcte. Ces preuves s'appuient généralement sur les méthodes déductives de vérification.

En pratique, une vérification efficace des programmes nécessite des outils tels que Why3 [28], l'Atelier B [26], ou encore la plateforme Rodin [3]. Ces outils qui sont eux-mêmes des programmes, peuvent se révéler d'une grande complexité, à la fois en ce qui concerne les principes mis en œuvre et en ce qui concerne leur implémentation. La vérification déductive requiert par exemple des procédures de décisions automatiques permettant de réduire l'effort de preuve. Dans le cas de la méthode B-classique, les principes sous-jacents sont formellement décrits dans un ouvrage de référence [2] de plus de 750 pages. Pourtant, la question de la confiance en ces principes s'est posée à propos de la composition des raffinements dans certains cas particuliers [66], et à propos de la correction de certains résultats intermédiaires [39]. Pour atteindre un niveau de confiance encore plus élevé, il faut donc aussi se pencher sur la question de la correction des outils de vérification et des principes qu'ils mettent en œuvre. En la matière les assistants de preuve représentent une sérieuse alternative dans la mesure où la question de la correction de ces outils est circonscrite à un petit noyau qu'il est humainement possible d'auditer, et sur lequel repose la correction de toute l'infrastructure proposée par l'assistant. Les assistants de preuve ont été conçus dans l'objectif d'aider à la spécification, à la conduite des raisonnements ainsi qu'à la vérification (mécanique) des raisonnements. De plus, les assistants de preuve basés sur la théorie des types mettent à la disposition du programmeur un langage fonctionnel typé d'une grande expressivité qui non seulement facilite la formalisation des spécifications et de la sémantique des langages, mais qui permet aussi de programmer et de certifier des outils tels que les générateurs d'obligations de preuve et les procédures de décision.

1.2 Contributions

Dans cette thèse nous étudions l'application de la démarche de programmation impérative par raffinements au sein de l'assistant de preuve *Coq*. Premièrement, nous formulons, dans le cadre de la théorie des types, une théorie de la programmation impérative par raffinements présentée à ICFEM 2019 [69]. Notre formulation est le résultat de la synthèse de divers points de vue tels que la programmation prédicative [33] de Hehner, ou encore les approches relationnelles du raffinement dues par exemple à Mili [50], Frappier [30], et Tchier [75]. Le point de vue de la programmation prédicative, qui consiste à considérer uni-

formément les programmes et les spécifications comme des prédicats, se révèle très adapté à une description de la sémantique relationnelle des programmes non-déterministes. Aussi, le point de vue relationnel nous a mené à une sémantique des boucles non-déterministes qui nous semble plus facile à appréhender que les sémantiques relationnelles classiques.

Deuxièmement, nous étudions une alternative à l'utilisation des notions classiques d'invariant et de variant pour raisonner sur les boucles. En pratique les spécifications associées aux corps des boucles ont intrinsèquement une partie relationnelle puisque la notion de décroissance du variant relie les valeurs de ce dernier avant et après l'exécution du corps de la boucle. Or, les approches relationnelles de la preuve de programme permettent de spécifier les corps des boucles de la même manière que n'importe quel programme, c'est-à-dire comme des relations entre leur pré-états et leur post-états. Partant de la technique de raisonnement sur les boucles non-déterministes due à Frappier [30], nous généralisons cette technique, et nous prouvons la complétude de notre généralisation.

Enfin, nous avons effectué un plongement de notre théorie dans le formalisme de l'assistant de preuve Coq. À la différence des travaux de Boulmé [13] et Alpuim et Swierstra [4] basés sur la théorie du raffinement de Morgan, notre approche améliore la lisibilité des développements notamment par l'utilisation du concept de bloc spécifié [32], et par la séparation entre les étapes de raffinements et les détails des preuves de correction. Le développement Coq [1] que nous avons réalisé dans le cadre de cette thèse, permet d'appliquer en pratique une démarche certifiée de programmation par raffinements au sein d'une infrastructure pensée pour le raisonnement logique et la preuve interactive. Ainsi, la partie de notre développement à laquelle on a besoin d'accorder sa confiance est relativement réduite, et en fait la base de confiance est circonscrite aux seuls éléments suivants : notre formulation de la sémantique des programmes impératifs, la définition de la relation de raffinement, et le noyau de vérification de l'assistant de preuve.

1.3 Plan

Le plan de cette thèse est le suivant. Dans les chapitres 2 et 3 nous posons le contexte technique dans lequel nos travaux ont eu lieu. D'abord, nous rappelons quelques notions de théorie des types en insistant sur la logique formelle engendrée par cette théorie. Ensuite, nous rappelons également les fondements de la formalisation du processus de raffinement aussi bien du point de vue relationnel, que du point de vue prédicatif.

Dans le chapitre 4, nous présentons le langage impératif considéré. Il s'agit d'un langage similaire au langage impératif *While* [61], avec affectations, sélections, séquences,

[1] https://github.com/bsall/thesis-dev

boucles et appels de sous-programmes. À ce langage, nous associons une sémantique prédicative et relationnelle à partir de laquelle nous posons une définition formelle du raffinement algorithmique de programmes. Nous établissons ensuite les propriétés générales de la relation de raffinement qui justifient l'approche de programmation par raffinements successifs. Enfin, nous étudions la relation entre d'une part la sémantique prédicative et relationnelle que nous avons choisie, et d'autre part une approche plus classique dans le style de la logique de Hoare. En particulier, nous montrons que les deux approches ont en théorie la même puissance puisque tout raffinement valide dans une approche l'est aussi dans l'autre.

Dans le chapitre 5, nous nous intéressons aux techniques de preuve permettant en pratique d'établir la validité des raffinements. En effet, les démonstrations directes à partir des définitions se révèlent particulièrement fastidieuses. Globalement, nous étudions donc les moyens de simplifier les formules logiques qui émanent des énoncés de raffinement, et dont nous devons établir la validité afin de prouver formellement la correction des dits raffinements. La simplification des formules logiques mentionnées passe en particulier par la formulation d'un calcul de la plus faible pré-spécification jouant ici le rôle du calcul de la plus faible précondition dans les approches plus classiques. Dans ce chapitre, nous nous penchons aussi sur le cas des boucles. Précisément, nous explorons une alternative à l'utilisation des notions d'invariant et de variant. Cette alternative est inspirée du point de vue relationnel sur les boucles et s'appuie sur une spécification relationnelle et abstraite des corps de boucle. Nous généralisons cette approche relationnelle de la correction des boucles, et nous montrons que notre généralisation est aussi bien correcte, qu'elle est complète relativement à l'expressivité du langage de spécifications.

Dans le chapitre 6, nous formalisons un langage de développement qui permet de capturer textuellement et hiérarchiquement les étapes de raffinements. Ensuite nous décrivons un système formel pour raisonner sur les développements. Ce système formel permet de garantir que les développements sont corrects par construction, et nous montrons que les propriétés de correction et de complétude que nous avons établies pour nos méthodes de preuve des raffinements individuels, s'étendent à notre système formel de raisonnement sur les développements globaux.

Avant de conclure, nous décrivons, dans le chapitre 7, la mécanisation en Coq, de la théorie du développement par raffinements que nous avons formalisée. Nous insistons en particulier sur les choix qui ont été motivés par les contraintes inhérentes au formalisme de l'assistant Coq. De plus nous détaillons comment en pratique l'infrastructure de l'assistant de preuve est mise à profit pour faciliter l'encodage des programmes et la gestion des obligations de preuves correspondant aux différentes étapes de raffinement.

Chapitre 2

Préliminaires

Dans ce chapitre nous posons le contexte technique de nos travaux. En guise de préliminaires, nous commençons donc par quelques rappels à propos de la théorie des types de Per Martin-Löf [49] qui est notre cadre formel de travail, et qui est à la base du calcul des constructions inductives (ou CIC [12]) sur lequel repose l'assistant de preuve Coq. Ensuite, nous discutons la représentation des ensembles et des relations dans le cadre de cette théorie des types. Enfin, nous rappelons quelques résultats classiques concernant l'existence et la représentation des points fixes des fonctions monotones.

2.1 Rappels de théorie des types

Les théories des types sont des cadres formels permettant de décrire des objets (appelés *types*) et de conduire des raisonnements formels sur ces objets. Ainsi les objets d'étude de ces théories sont les types et tous les objets ont chacun un type (et un seul) qui par conséquent représente la nature intrinsèque des objets d'une collection donnée d'objets. En guise de point de départ pour la construction de nouveaux types, nous acceptons les faits suivants :

1. il existe un type désigné par Type_0
2. pour tout entier naturel i, si Type_i est un type, alors Type_{i+1} est aussi un type

Cette hiérarchisation des types permet de prémunir la théorie des paradoxes bien connus de la théorie naïve des ensembles. Nous considérerons de plus que la hiérarchie des types ainsi définie est cumulative, soit:

3. pour tout objet T, si T a pour type Type_i alors T a aussi pour type Type_{i+1}

Types et propositions logiques. La théorie des types permet de conduire des raisonnements logiques lorsque les types sont interprétés comme des propositions. Dans cette interprétation, on considère qu'une proposition P est vraie si et seulement si il existe au moins un terme e tel que e est de type P. On dit que P est habité. Donc, à la différence de la logique classique dans laquelle on admet qu'une proposition peut être vraie sans qu'une preuve de cette dernière existe, en théorie des type la validité d'un énoncé correspond exactement à l'existence d'une preuve. Ainsi, la proposition Faux (notée $\bot$) correspond au type vide qui n'est habité par aucun terme. Et la proposition Vrai (notée $\top$) correspond au type ayant un unique habitant.

Type produit et quantification universelle. Une première manière de composer des types existants pour former de nouveaux types consiste à appliquer la règle d'introduction suivante. Pour tout type T et tout type T', on peut former un type produit comme suit :

$$\frac{T : \mathsf{Type},\ T' : \mathsf{Type}}{\prod_{(x:T)} T' \ : \ \mathsf{Type}}$$

Il s'agit du type des fonctions totales de T vers $T'(x)$, c'est-à-dire, des termes de la forme $(\lambda(x : T) \Rightarrow E(x))$ où $E(x)$ de type T' et x est de type T. Dans le cas où T' ne dépend pas de l'entrée x, le type produit est également noté comme suit :

$$\prod_{(x:T)} T' \quad \equiv \quad \prod_{(\_:T)} T' \quad \equiv \quad T \to T'$$

Logiquement parlant, le type $T \to T'$ s'interprète comme une implication dans la mesure où les termes de ce type permettent de construire un habitant de T' si on dispose d'un habitant de T. Autrement dit, dire que la proposition $T \to T'$ est vraie signifie que, si la proposition T est vraie, alors la proposition T' est vraie. Par généralisation, le type produit $(\prod_{(x:T)} T')$ peut s'interpréter logiquement comme suit : pour tout x de type T, on sait construire à partir de x un habitant du type T'. En conséquence, selon l'interprétation des types comme proposition, on peut dire que pour tout x, T' est vrai. D'où la notation usuelle suivante :

$$\prod_{(x:T)} T' \quad \equiv \quad \forall\ (x : T) \cdot T'$$

Dans le cadre de la théorie des types, prouver la validité d'une proposition de la forme $\forall\ (x : T) \cdot T'$ va donc consister à construire un terme de la forme $(\lambda\ x \Rightarrow E(x))$ dans lequel l'expression $E(x)$ a le type T'.

Type somme et quantification existentielle. Soient T et T' deux types, alors nous pouvons former un type somme comme suit:

$$\frac{T : \mathsf{Type},\ T' : \mathsf{Type}}{\sum_{(x:T)}, T' \ : \ \mathsf{Type}}$$

Lorsque T' ne dépend pas de l'entrée x, ce type se note $T \times T'$, il s'agit du type des couples d'éléments dont le premier élément est de type T et le second élément est de type T'. En particulier, si e est de type T et e' de type T', alors (e, e') est de type $(T \times T') \equiv (\sum_{(\_:T)}, T')$. De plus, Si on considère T et T' comme des propositions alors le couple (e, e') peut s'interpréter comme une preuve de la proposition $T \wedge T'$, c'est à dire :

$$T \wedge T' \quad \equiv \quad T \times T'$$

En effet dès lors qu'on dispose du couple (e, e') avec e preuve de T et e' preuve de T', on sait que la proposition $T \wedge T'$ est vraie puisque habitée par (e, e'). Et si (e, e') habite de $T \wedge T'$ alors, T est vraie puisque habitée par e, et T' est vraie puisque habitée par e'.

Lorsque T' dépend de x, $\sum_{(x:T)} T'$ représente le type des couples formés par un élément quelconque x de type T et un élément quelconque x' de type T'. Ainsi tout habitant de ce type est une preuve qu'il existe x et x' tels que x est de type T et T' est habité par x'. C'est-à-dire qu'il existe x de type T tel que la proposition T' est vraie, les $\sum$-types permettent donc de représenter la quantification existentielle logique :

$$\sum_{(x:T)} (P\ x) \quad \equiv \quad \exists\ (x : T) \cdot (P\ x) \tag{2.1}$$

De là, il est facile de voir que les $\sum$-types permettent aussi de représenter la disjonction logique. En effet, considérons la proposition suivante :

$$P \stackrel{\text{def}}{=} \lambda\ (x : \mathsf{bool}) \Rightarrow \textbf{if}\ x\ \textbf{then}\ T_1\ \textbf{else}\ T_2\ \textbf{end}$$

alors on a $P\ \mathsf{false} = T_2$ et $P\ \mathsf{true} = T_1$. Donc, une preuve de $T_1 \vee T_2$ est une preuve de $(P\ \mathsf{true}) \vee (P\ \mathsf{false})$, c'est-à-dire une preuve de $\exists\ (x : \mathsf{bool}) \cdot P$, soit, selon 2.1, une preuve de $\sum_{(x:\mathsf{bool})} P$. Par conséquent on a la représentation suivante de la disjonction logique :

$$T_1 \vee T_2 \quad \equiv \quad \sum_{(x:\mathsf{bool})} (\textbf{if}\ x\ \textbf{then}\ T_1\ \textbf{else}\ T_2\ \textbf{end})$$

De manière analogue au type produit, prouver la validité d'une proposition de la forme

$\sum_{(x:T)} T'$ va consister à fournir un terme x de type T et un terme t de type T'.

Négation. Si supposer qu'une proposition T est vraie conduit à une absurdité, alors on peut considérer que T est en réalité fausse. Une autre formulation en théorie des types consiste à dire que T est faux signifie qu'à partir d'une preuve de T, on sait construire une preuve de $\bot$. On définira donc la négation d'une proposition T par le type des fonctions de T vers $\bot$:

$$\neg\ T \ \stackrel{\text{def}}{=}\ T \to \bot$$

À la différence de la logique classique, la proposition $(\neg\neg\ T \to T)$ n'est pas un théorème dans la logique intuitionniste engendrée par la théorie des types. Cette proposition est d'ailleurs la seule règle de la déduction naturelle qui n'est pas valide en logique intuitionniste. Par conséquent, pour montrer que T est vrai on ne pourra pas raisonner par l'absurde en général. Néanmoins le raisonnement par l'absurde reste possible lorsque T est de la forme $\neg T'$, puisque dans ce cas il suffit de déduire le faux après avoir supposé la validité de T'. Les propositions $(\neg(P \wedge Q) \to \neg P \vee \neg Q)$, $((P \to Q) \to \neg P \vee Q)$, ainsi que $(\neg(\forall\ x \cdot P\ x) \to (\exists\ x \cdot \neg P\ x))$ ne sont pas des théorèmes non plus.

Décidabilité. La notion de décidabilité est fondamentale en programmation. Il est donc important pour une théorie de la programmation de pouvoir rendre explicite les hypothèses de décidabilité. En logique classique la validité du *tiers exclu* permet de poser des hypothèses implicites de décidabilité pouvant conduire à des programmes impossible à implémenter lorsque lesdites hypothèses ne sont pas valides. Pour éviter ce problème if faut raisonner en étant en permanence sur ses gardes, ce qui est d'autant plus difficile que notre familiarité avec la logique classique est grande. A la différence de la logique classique, la théorie des types engendre une logique intuitionniste qui oblige à rendre les hypothèses de décidabilité explicites du fait de la non validité du tiers exclu dans cette logique.

Définition 1. *On dira qu'une proposition P est décidable si et seulement si $P \vee \neg P$ est valide. Formellement, on fera référence à la décidabilité de P par le prédicat* decidable P *défini comme suit :*

$$\begin{array}{lcl} \mathit{decidable} & : & \mathit{Type} \to \mathit{Type} \\ \mathit{decicable} & \stackrel{\text{def}}{=} & \lambda\ P \Rightarrow P \vee \neg P \end{array}$$

Calcul et récursion. Étant basée sur le λ-calcul, la théorie des types intègre une notion native de calcul. Cependant quelques restrictions supplémentaires sont nécessaires. En

effet, considérons la fonction absurd définie comme suit :

$$\mathsf{absurd}\ (x : T) : \bot\ :=\ \mathsf{absurd}\ x$$

Si nous laissons de côté la question de la terminaison, cette définition est bien typée. Par contre, elle n'est pas acceptable puisqu'elle permet de construire une preuve de $\bot$. En effet, s'il existe un terme x de type T, alors le terme ($\mathsf{absurd}\ x$) est de type $\bot$. La logique sous-jacente serait alors incohérente. Par conséquent, une fonction ne peut être considérée comme étant bien typée si elle n'est pas totale.

Égalité. On considérera que la notion d'égalité est représentée par le type suivant des égalités sur un type T donné :

$$eq\ (T : \mathsf{Type})(x : T)(y : T) : \mathsf{Type}$$

Le type eq est paramétré par un type T et par les deux éléments x et y qui sont impliqués dans la relation d'égalité. On considérera aussi que ce type possède l'unique constructeur par réflexivité refl défini comme suit :

$$\mathsf{refl} : \forall\ (T : \mathsf{Type})(x : T) \cdot \mathsf{eq}\ T\ x\ x$$

Ainsi, pour tout x de type T, il est toujours possible de construire une preuve de $x = x$ (c'est-à-dire un habitant de $\mathsf{eq}\ T\ x\ x$) en appliquant le constructeur refl aux arguments T et x. De plus, comme refl est le seul moyen de construire des preuves d'égalités, la proposition $x = y$ reflète bien le fait que x et y sont identiques (modulo β-réduction), donc interchangeables. À partir de cette caractérisation de l'égalité, on peut montrer que $\mathsf{eq}\ T$ est une relation d'équivalence. On peut aussi montrer que deux termes égaux sont effectivement interchangeables, c'est-à-dire :

$$\forall\ (T : \mathsf{Type})(P : T \to \mathsf{Type})(x\ y : T) \cdot P\ x\ \to\ x = y\ \to\ P\ y$$

Autrement dit, si $x = y$, alors, toute preuve de la proposition ($P\ x$) est convertible en une preuve de ($P\ y$).

2.2 Ensembles et relations en théorie des types

Dans cette section, nous présentons une codification classique des ensembles, des relations, ainsi que de quelques autres notions associées, dans le cadre de la théorie des types. Cette codification nous sera utile notamment pour parler formellement de la syntaxe et de la sémantique des langages impératifs.

Codification des ensembles

Un type T peut être interprété comme un ensemble d'objets. Il se pose alors la question de la représentation des sous-ensembles de T. Il serait peu commode de devoir déclarer un nouveau type à chaque fois qu'on souhaite parler d'un sous-ensemble. Une codification plus appropriée des ensembles consiste à utiliser systématiquement la définition par compréhension qui permet de donner une représentation ensembliste à tout type T ainsi qu'à ses sous-ensembles.

Définition 2 (Sous-ensembles de type)**.** *Soit T un type. Les ensembles d'objets de type T (noté Set_T) sont les habitants de $T \to \mathsf{Type}$:*

$$\mathsf{Set}_T \stackrel{\text{def}}{=} T \to \mathsf{Type}$$

Si $\mathcal{P}$ est un prédicat sur un objet t de type T, la définition 2 revient à dire que l'ensemble des objets t tel que $\mathcal{P}$ est vrai correspond à la fonction qui à un objet t associe la proposition $\mathcal{P}$:

$$(\lambda\ (t : T) \Rightarrow \mathcal{P}) \ \approx \text{ l'ensemble des objets } t \text{ tels que } \mathcal{P} \text{ est vrai}$$

Exemple 1.
$(\lambda\ (t : T) \Rightarrow \mathsf{False})$ *représente l'ensemble vide ϕ.*
$(\lambda\ (t : T) \Rightarrow \mathsf{True})$ *représente l'ensemble des objets de type T.*
$(\lambda\ (n : \mathsf{nat}) \Rightarrow n > 10)$ *représente l'ensemble des entiers plus grand que 10*

Notons la représentation de l'ensemble E_i par $F_{E_i} = \lambda\ (t : T) \Rightarrow \mathcal{E}_i$ de type Set_T. Alors, l'appartenance à E_i d'un élément x correspond à la validité de $(F_{E_i}\ x)$:

$$x \in E_i \ \approx \ ((F_{E_i}\ x) \text{ est habité}) \text{ ou } (\mathcal{E}_{i[x/t]} \text{ est habité})$$

L'union, l'intersection et la complémentation de E sont naturellement représentées par la

disjonction, la conjonction et la négation logique :

$$\begin{array}{lcl} E_1 \cup E_2 & \stackrel{\text{def}}{=} & \lambda\ (t : T) \Rightarrow F_{E_1}\ t\ \vee\ F_{E_2}\ t \\ E_1 \cap E_2 & \stackrel{\text{def}}{=} & \lambda\ (t : T) \Rightarrow F_{E_1}\ t\ \wedge\ F_{E_2}\ t \\ \overline{E_i} & \stackrel{\text{def}}{=} & \lambda\ (t : T) \Rightarrow \neg F_{E_i}\ t \end{array}$$

L'inclusion ensembliste correspond à une implication logique généralisée. On considérera que E_1 est incluse dans E_2, si de toute preuve de $(F_{E_1}\ t)$, on peut déduire une preuve de $(F_{E_2}\ t)$, quel que soit t. De même, la double inclusion, que nous noterons par $(\equiv)$ correspond à une équivalence logique généralisée. Par conséquent, on pose les définitions suivantes :

$$\begin{array}{lcl} E_1 \subseteq E_2 & \stackrel{\text{def}}{=} & \forall\ t \cdot F_{E_1}\ t \rightarrow F_{E_2}\ t \\ E_1 \equiv E_2 & \stackrel{\text{def}}{=} & \forall\ t \cdot F_{E_1}\ t \leftrightarrow F_{E_2}\ t \end{array}$$

Codification des relations

La codification d'une relation comme l'ensemble des paires d'éléments que la relation associe se déduit aisément de la codification des ensembles. Les relations d'un ensemble Set_T vers Set_U peuvent donc être représentées par $\mathsf{Set}_{T \times U}$. Une autre représentation isomorphe à cette dernière est celle qui définit les relations de Set_T vers Set_U comme les objets de type $T \rightarrow U \rightarrow \mathsf{Type}$. Nous préférerons cette seconde représentation qui est plus directe dans la mesure où elle ne fait pas intervenir la notion de couple.

Définition 3 (Relations)**.** *Soient T et U deux types. Les relations hétérogènes de Set_T vers Set_U (noté $\mathsf{Rel}_{T,U}$) sont les habitants de $T \rightarrow U \rightarrow \mathsf{Type}$:*

$$\mathsf{Rel}_{T,U} \stackrel{\text{def}}{=} T \rightarrow U \rightarrow \mathsf{Type}$$

Exemple 2. *L'addition d'entiers a la représentation relationnelle suivante :*

$$\begin{array}{lcl} \mathsf{radd} & : & \mathsf{nat} \times \mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{Type} \\ \mathsf{radd} & \stackrel{\text{def}}{=} & \lambda\ (x, y)\ z \Rightarrow x + y = z \end{array}$$

La codification des notions habituelles de domaine, codomaine et composition est donnée par les définitions ci-dessous.

Définition 4 (Domaine et co-domaine)**.** *Soient les types T et U, et R de type $\mathsf{Rel}_{T,U}$. Le domaine de R est défini comme suit :*

$$\begin{array}{lcl} \mathsf{dom} & : & \mathsf{Rel}_{T,U} \rightarrow \mathsf{Set}_T \\ \mathsf{dom} & \stackrel{\text{def}}{=} & \lambda\ R \Rightarrow \lambda\ t \Rightarrow \exists\ u \cdot R\ t\ u \end{array}$$

De manière analogue, le codomaine de R est défini comme suit :

$$\begin{array}{lcl} \mathsf{cod} & : & \mathsf{Rel}_{T,U} \to \mathsf{Set}_U \\ \mathsf{cod} & \stackrel{\text{def}}{=} & \lambda\ R \Rightarrow \lambda\ u \Rightarrow \exists\ t \cdot R\ t\ u \end{array}$$

Définition 5 (Composition de relations)**.** *Soient les types T, U et V. Soient R_1 et R_2 deux relations de types respectifs $\mathsf{Rel}_{T,U}$ et $\mathsf{Rel}_{U,V}$. On définit comme suit la composition $R_1 \circ R_2$ de R_1 et R_2 :*

$$\begin{array}{lcl} R_1 \circ R_2 & : & \mathsf{Rel}_{T,V} \\ R_1 \circ R_2 & \stackrel{\text{def}}{=} & \lambda\ t\ v \Rightarrow \exists\ (u : U) \cdot R_1\ t\ u\ \wedge\ R_2\ u\ v \end{array}$$

La notion de relation bien fondée est très utile, notamment pour raisonner sur la terminaison des boucles. Ici, nous définirons cette notion par le principe d'induction qui lui est associé.

Définition 6 (Relation bien fondée)**.** *Soit T un type, et soit R de type $\mathsf{Rel}_{T,T}$. On dira que R est bien fondée si et seulement si le prédicat $(\mathsf{wfd}\ R)$ est valide, avec*

$$\begin{array}{lcl} \mathsf{wfd} & : & \mathsf{Rel}_{T,T} \to \mathsf{Type} \\ \mathsf{wfd} & \stackrel{\text{def}}{=} & \lambda\ R \Rightarrow \forall\ (P : \mathsf{Set}_T) \cdot (\forall\ t \cdot (\forall\ s \cdot R\ s\ t \to P\ t) \to P\ t)) \to (\forall\ t \cdot P\ t) \end{array}$$

Exemple 3. *Considérons la relation d'infériorité sur les entiers représentée par la définition suivante :*

$$R \stackrel{\text{def}}{=} \lambda\ (n\ n' : \mathsf{nat}) \Rightarrow n < n'$$

Cette relation est bien fondée, et $\mathsf{wfd}\ R$ correspond bien au principe d'induction complète sur les entiers :

$$\mathsf{wfd}\ R \stackrel{\text{def}}{=} \forall\ (P : \mathsf{Set}_{\mathsf{nat}}) \cdot (\forall\ n' \cdot (\forall\ n \cdot n < n' \to P\ n) \to P\ n')) \to (\forall\ n \cdot P\ n)$$

2.3 Points fixes et fermetures transitives

Dans cette section, nous nous intéressons à la codification des notions de point fixe et de fermeture transitive. Notamment, ces notions sont utiles à la définition de la sémantique des boucles et au raisonnement sur les boucles. En particulier nous nous intéressons aux points fixes des fonctions monotones. Commençons donc par rappeler la définition d'une fonction monotone.

Définition 7 (Fonction monotone)**.** *Soit T un type, et soit la fonction $F : \mathit{Set}_T \to \mathit{Set}_T$. On définit la monotonie de F par le prédicat $\mathit{monotonic}\ F$ défini comme suit :*

$$\begin{array}{lcl} \mathit{monotonic} & : & (\mathit{Set}_T \to \mathit{Set}_T) \to \mathit{Type} \\ \mathit{monotonic} & \stackrel{\text{def}}{=} & \lambda\ F \Rightarrow \forall\ X\ Y \cdot X \subseteq Y\ \to\ (F\ X) \subseteq (F\ Y) \end{array}$$

Nous définissons maintenant un opérateur de point fixe qui nous permet de formaliser, s'il existe, le plus petit point fixe d'une fonction monotone.

Définition 8 (Plus petit point fixe)**.** *Soit T un type, et soit F une fonction de type $\mathit{Set}_T \to \mathit{Set}_T$. On définit l'opérateur de plus petit point fixe lfp comme suit :*

$$\begin{array}{lcl} \mathit{lfp} & : & (\mathit{Set}_T \to \mathit{Set}_T) \to \mathit{Set}_T \\ \mathit{lfp} & \stackrel{\text{def}}{=} & \lambda\ F \Rightarrow \lambda\ t \Rightarrow \forall\ X \cdot (F\ X \subseteq X) \to X\ t \end{array}$$

En termes ensemblistes, $\mathsf{lfp}\ F$ correspond à l'intersection de tous les ensembles X tels que $F\ X \subseteq X$. C'est-à-dire :

$$\mathsf{lfp}\ F\ \equiv\ \bigcap \{\ X \mid F\ X \subseteq X\}$$

Comme l'a montré Tarski, $\mathsf{lfp}\ F$ a les deux propriétés suivantes.

Lemme 1 (F fixe ($\mathsf{lfp}\ F$) [74])**.** *Soit T un type, et soit la fonction F de type $\mathit{Set}_T \to \mathit{Set}_T$. Alors si F est monotone, $\mathit{lfp}\ F$ est un point fixe de F :*

$$\mathit{monotonic}\ F\quad \to\quad F\ (\mathit{lfp}\ F) \equiv \mathit{lfp}\ F$$

Lemme 2 (($\mathsf{lfp}\ F$) est minimal [74])**.** *Soit T un type, et soit la fonction $F : \mathit{Set}_T \to \mathit{Set}_T$. Alors, si F est monotone, $\mathit{lfp}\ F$ est le plus petit point fixe de F :*

$$\mathit{monotonic}\ F\quad \to\quad \forall\ X \cdot (F\ X \equiv X)\ \to\ (\mathit{lfp}\ F) \subseteq X$$

La notion de point fixe permet de définir la fermeture transitive d'une relation R comme le plus petit point fixe d'une fonction monotone. La définition que nous utiliserons est donnée ci-dessous.

Définition 9 (Fermeture transitive)**.** *Soit le type T, et soit la relation R de type $\mathit{Rel}_{T,T}$. On définit la fermeture transitive R^+ de R comme suit :*

$$\begin{array}{lcl} \_^+ & : & \mathit{Rel}_{T,T} \to \mathit{Rel}_{T,T} \\ R^+ & \stackrel{\text{def}}{=} & \mathit{lfp}\ (\lambda\ X \Rightarrow R \circ X\ \cup\ R) \end{array}$$

À partir des propriétés des points fixes énoncées précédemment, il est possible de montrer que cette définition de la fermeture transitive a les deux propriétés suivantes qui en font la plus petite relation transitive contenant R.

Lemme 3 (R^+ est transitive)**.** *Soit le type T, et soit la relation R de type $\mathsf{Rel}_{T,T}$. Alors, R^+ est transitive :*

$$R^+ \circ R^+ \subseteq R^+$$

Lemme 4 (R^+ est minimale)**.** *Soit le type T, et soit la relation R de type $\mathsf{Rel}_{T,T}$. Alors, toute relation transitive R' contenant R contient aussi R^+ :*

$$\forall\ R' \cdot R' \circ R' \subseteq R' \ \rightarrow\ R \subseteq R' \ \rightarrow\ R^+ \subseteq R'$$

Chapitre 3

État de l'art

Le *raffinement pas-à-pas*[1] a été introduit par Dijkstra [24] et Wirth [80] comme technique de développement de programmes informatiques. Cette technique consiste à développer un programme par étapes successives, en partant des spécifications pour aboutir à un programme final correct par construction (c'est-à-dire qui satisfait aux spécifications initiales). L'introduction du raffinement est motivée par la difficulté de la démarche inverse qui consiste à prouver a posteriori qu'un programme satisfait aux spécifications, en général sans connaître les raisons ayant motivé tel choix de conception plutôt que tel autre. Au contraire, avec la technique de raffinement, les étapes successives de raffinement peuvent être archivées, ce qui permet de comprendre le cheminement ayant abouti au programme final. Le passage d'une étape de raffinement à la suivante fait l'objet d'une justification permettant de garantir la correction de chaque raffinement, la correction du programme final en regard de la spécification initiale découle par transitivité de la correction des raffinements successifs. Initialement, les spécifications et les justifications sont informelles, il n'est donc pas aisé de s'assurer que les justifications sont des preuves correctes. Ce problème est approché de deux manières. On a d'un côté un traitement formel du raffinement dans la théorie des relations. Et d'un autre côté on a une approche prédicative qui intègre les spécifications formelles dans les langages sous forme de *pré* et *post* conditions. Il devient alors possible de s'appuyer sur la théorie des relations, sur la logique de Hoare [35] ou encore sur le calcul de la plus faible précondition [25], pour prouver formellement la correction des raffinements.

[1] de l'anglais *stepwise refinement*

3.1 L'approche relationnelle

L'utilisation des approches relationnelles semble avoir commencé avec Mills qui dans [52] décrit les programmes déterministes par des fonctions qui associent entrées et sorties de programmes. La sémantique d'un programme P est donnée par une fonction (possiblement partielle) f sur l'espace E des états de P avec l'interprétation suivante : P exécuté dans l'état initial e de E termine sur l'état final e' de E, si et seulement si e appartient au domaine de f et $e' = (f\ e)$. Pour désigner la sémantique de P on utilisera la notation $[\![P]\!]$. Si la sémantique de P est la fonction f, alors $[\![P]\!]$ est définie par comme suit :

$$[\![P]\!] \stackrel{\text{def}}{=} \lambda\ e\ e' \Rightarrow e' = (f\ e)$$

Considérant que chaque instruction de base d'un langage dénote une fonction de base donnée, Mills donne par induction une caractérisation fonctionnelle de l'alternative et de la composition séquentielle. L'itération quant à elle est caractérisée par une expansion récursive.

$$\begin{array}{lcl}
[\![\textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}]\!] & \stackrel{\text{def}}{=} & \lambda\ e\ e' \Rightarrow ((C\ e) \wedge [\![S_1]\!]\ e\ e')\ \vee\ (\neg(C\ e) \wedge [\![S_2]\!]\ e\ e') \\
[\![S_1 \textbf{ ; } S_2]\!] & \stackrel{\text{def}}{=} & \lambda\ e\ e' \Rightarrow \exists\ e_x \cdot [\![S_1]\!]\ e\ e_x\ \wedge\ [\![S_2]\!]\ e_x\ e' \\
[\![\textbf{while } C \textbf{ do } S \textbf{ end}]\!] & \equiv & [\![\textbf{if } C \textbf{ then } (S \textbf{ ; while } C \textbf{ do } S \textbf{ end}) \textbf{ end}]\!]
\end{array}$$

Mills donne aussi les conditions nécessaires et suffisantes qui permettent de montrer qu'une fonction f donnée dénote bien un programme P. Globalement, il faut et il suffit de montrer que

$$\forall\ e\ e' \cdot e' = (f\ e)\ \leftrightarrow\ [\![P]\!]\ e\ e'$$

Il y a néanmoins une difficulté avec les boucles car leur sémantique n'est pas calculable en général. Néanmoins grâce à la *règle de vérification des boucles de Mills*, il est possible de vérifier qu'une fonction f correspond bien à la sémantique d'une boucle $\mathcal{B} \stackrel{\text{def}}{=} \textbf{while } C \textbf{ do } S \textbf{ end}$. Cette règle énonce que pour ce faire il faut et il suffit de montrer que :

$$\forall\ e\ e' \cdot e' = (f\ e)\ \ \rightarrow\ \ \mathcal{B} \text{ termine } \wedge\ ((C\ e) \wedge \exists e_x \cdot [\![S]\!]\ e\ e_x \wedge (e' = f\ e_x)\ \vee\ \neg(C\ e) \wedge e' = e)$$

Comme Mills et Linger l'explique plus en détail dans [54] et [47], ces conditions sont équivalentes aux deux conditions suivantes :

$$\begin{array}{l}
(\mathsf{dom}\ f) = (\mathsf{dom}\ [\![\textbf{while } C \textbf{ do } S \textbf{ end}]\!]) \\
f \text{ est un point fixe de } (\lambda\ g \Rightarrow \textbf{if } C \textbf{ then } g \circ S \textbf{ end})
\end{array}$$

La démarche de Mills aboutit à une sémantique fonctionnelle et intuitive des programmes ainsi qu'à des techniques de preuves abordables même si on aurait pu espérer des conditions suffisantes plus pratiques permettant de prouver la terminaison des boucles. En revanche, elle se limite aux programmes déterministes et aux spécifications déterministes. De plus, elle délègue implicitement la question des preuves de terminaison à d'autres formalismes ou techniques de preuves comme par exemple l'identification d'une mesure sur un ensemble bien fondé.

Dans [50], Mili généralise les travaux de Mills aux spécifications non déterministes. Ces travaux de Mili concernent les programmes déterministes, mais semblent déjà pouvoir s'étendre aux programmes non déterministes. Dès lors qu'une spécification est non déterministe, elle ne peut correspondre exactement à un programme déterministe, il faut donc une notion de correction plus faible que la correspondance exacte. Mili défini la correction d'un programme P vis-à-vis d'une spécification relationnelle R comme suit :

$$P \sqsubseteq R \quad \stackrel{\text{def}}{=} \quad \forall\, e \cdot \mathsf{dom}\ R\ e \rightarrow (\mathsf{dom}\ [\![P]\!]\ e \quad \wedge \quad \forall\, e' \cdot [\![P]\!]\ e\ e' \rightarrow R\ e\ e') \tag{3.1}$$

Dans cette définition, R est une relation possiblement non déterministe et $[\![P]\!]$ est une fonction, donc une relation déterministe. Mili relie cette définition à une définition équivalente et plus concise qui avait été donnée par Mills dans [53], soit :

$$P \sqsubseteq R \quad \stackrel{\text{def}}{=} \quad \mathsf{dom}\ (R \cap [\![P]\!]) \ \equiv\ \mathsf{dom}\ R$$

Dans ces travaux Mili s'est surtout intéressé à une démarche de dérivation de programmes à partir des spécifications. Cependant, la règle correspondante de dérivation pour les boucles combinée à la règle de la conséquence peuvent permettre de vérifier qu'une boucle implémente bien une spécification relationnelle donnée. Quand on considère les programmes non-déterministes, la définition 3.1 reste appropriée. En effet, dans [51], Mili et Desharnais, définissent en termes très généraux la notion de raffinement entre deux relations binaires comme suit.

$$R_2 \sqsubseteq R_1 \quad \stackrel{\text{def}}{=} \quad \forall\, e \cdot \mathsf{dom}\ R_1\ e \rightarrow \mathsf{dom}\ R_2\ e \quad \wedge \quad \forall\, e' \cdot R_2\ e\ e' \rightarrow R_1\ e\ e' \tag{3.2}$$

Pour tenir compte du non déterminisme des programmes, il est nécessaire de réexaminer l'interprétation de la composition séquentielle puisque la composition relationnelle n'est pas monotone vis-à-vis de la relation de raffinement, et de plus, elle ne rend plus correctement compte du comportement de la séquence selon l'interprétation relationnelle des

programmes. Par exemple, la notion de composition démoniaque [9], qui est une restriction de la composition relationnelle, est utilisée dans les travaux de Desharnais [23], Frappier [30] et Tchier [75] pour définir la sémantique formelle de la composition séquentielle des programmes.

Dans le formalisme de la notation Z [73], un état spécial noté $\perp$ permet de représenter la non-terminaison. Dans ce cas, l'interprétation est que le programme S est susceptible de ne pas terminer sur l'entrée i si $(i, \perp) \in S$. Une autre manière de représenter la non-terminaison consiste à dire que S ne termine pas sur i si $(\forall\ o \cdot (i, o) \in S)$, cette interprétation correspond notamment à la sémantique relationnelle utilisée dans la méthode B [2]. La relation de raffinement associée à ces interprétations se formalise généralement comme suit. Soient S et S' deux programmes, alors on a la définition formelle suivante :

$$S' \text{ raffine } S \quad \stackrel{\text{def}}{=} \quad S' \subseteq S \wedge (\mathsf{dom}\ S) \subseteq (\mathsf{dom}\ S')$$

Autrement dit, si on considère les comportements observables de S et S', alors S' raffine S si et seulement si tout comportement observable de S' est aussi un comportement observable de S : $S' \subseteq S$, et S' termine à chaque fois que S termine : $(\mathsf{dom}\ S) \subseteq (\mathsf{dom}\ S')$.

Avec l'approche relationnelle on dispose d'un formalisme permettant d'exprimer les spécifications et les programmes. La notion de raffinement est prise en compte dans cette approche, ce qui permet l'application de la démarche de développement par raffinements successifs. Néanmoins, bien que la théorie des relations soit suffisamment expressive, la formalisation des spécifications et des programmes dans cette théorie introduit un niveau d'indirection qui est le formalisme de la théorie des ensembles. A contrario, les approches prédicatives suppriment ce niveau d'indirection en permettant une expression plus directe des programmes et des spécifications dans le langage de la logique.

3.2 Les approches prédicatives

Les approches prédicatives du raffinement définissent le raffinement directement dans le calcul des prédicats. La logique de Hoare et le calcul de la plus faible précondition (wp-calcul [25]) sont alors les principaux outils permettant d'effectuer des preuves formelles de raffinement.

3.2.1 Logique de Hoare.

Dans l'approche prédicative, on dit qu'un programme S' raffine un programme S si et seulement si toute spécification satisfaite par S est aussi satisfaite par S', étant entendu que les spécifications sont formalisées sous la forme d'une paire de prédicats. On en déduit la définition suivante du raffinement en logique de Hoare :

$$S' \sqsubseteq S \quad \stackrel{\text{def}}{=} \quad \forall\, P\ Q \cdot [P]\ S\ [Q] \ \rightarrow\ [P]\ S'\ [Q]$$

3.2.2 Calcul de la plus faible précondition.

De manière générale, prouver les triplets de Hoare nécessite d'appliquer les règles d'inférence de la logique sous-jacente, et l'application de ces règles d'inférence nécessite parfois la fourniture de prédicats intermédiaires. Par exemple, pour démontrer $[P]\ S;T\ [Q]$ il faut trouver R tel que $[P]\ S\ [R] \wedge [R]\ T\ [Q]$. Ceci représente un frein à la mécanisation des preuves. Le wp-calcul permet d'améliorer la situation dans la mesure où il permet de transformer le problème de la preuve d'un programme en un problème de validité d'un prédicat. Le wp-calcul et la logique de Hoare ont la même puissance :

$$\forall\, P\ Q \cdot [P]\ S\ [Q] \leftrightarrow (P \rightarrow (\mathsf{wp}\ S\ Q))$$

On peut donc en toute généralité définir le raffinement en wp-calcul :

$$S' \sqsubseteq S \quad \stackrel{\text{def}}{=} \quad \forall\, Q \cdot (\mathsf{wp}\ S\ Q) \rightarrow (\mathsf{wp}\ S'\ Q)$$

Comme $\mathsf{wp}\ (S;T)\ Q = \mathsf{wp}\ S\ (\mathsf{wp}\ T\ Q)$, pour démontrer $[P]\ S;T\ [Q]$, il suffit de monter que $P \rightarrow (\mathsf{wp}\ (S;T)\ Q)$, c'est-à-dire $P \rightarrow (\mathsf{wp}\ S\ (\mathsf{wp}\ T\ Q))$. Le prédicat intermédiaire R n'est plus nécessaire, ce qui favorise la mécanisation de certaines étapes de raisonnement. Néanmoins, la quantification sur le prédicat p rend cette définition difficile à utiliser en pratique. Il est intéressant de noter la relation suivante entre la logique de Hoare et le calcul de la plus faible précondition :

$$\mathsf{wp}\ S\ Q \ \equiv\ \lambda\, e \Rightarrow \exists\, X \cdot [X]\ S\ [Q] \ \wedge\ X\ e$$

Cette relation décrit la plus faible précondition pour que S établisse la postcondition Q comme l'union (ou la somme) de toutes les préconditions telles ques S établit Q. Il est facile de voir que pour toute précondition Y tel que le triplet $[Y]\ S\ [Q]$ est valide, on a bien $Y \subseteq (\mathsf{wp}\ S\ Q)$.

3.2.3 Programmation prédicative.

La démarche de Hehner consiste à unifier langage de spécification et langage de programmation, ce qui différencie cette approche d'autres approches comme B qui font la distinction entre prédicats (à prouver) et expressions booléennes (à évaluer), ou encore la logique de Hoare qui utilise une paire de prédicats tout comme l'instruction de spécification introduite par Morgan. La démarche de Hehner est détaillée dans [33] et résumée dans [31], nous donnons ici une synthèse issue de ces travaux.

Prédicats et Spécifications Dans la théorie de Hehner, comme en Z et VDM [40], les spécifications sont des prédicats *avant-après*, dans lesquelles on désigne par v' la valeur d'une variable v après l'exécution afin d'éviter le recours aux variables fantômes. De plus Hehner introduit une notion explicite de terminaison dans les spécifications. Si une variable h représente la terminaison de l'exécution, alors $(P \rightarrow h \wedge Q)$ est une spécification de correction totale de P vis-à-vis de la spécification Q, et $(P \wedge h \rightarrow Q)$ est une spécification de correction partielle.

L'utilisation des prédicats pour spécifier est justifiée par le fait que pour tout autre formalisme de spécification, il faudra d'une manière ou d'une autre dire ce que cela signifie de satisfaire une spécification exprimée dans ce formalisme. Le langage idéal pour ce faire reste la logique formelle (les prédicats). Par conséquent autant en rester aux prédicats. Par exemple, supposons que l'exécution d'un programme P démarre dans l'état $v = a$ et se termine dans l'état $v = b$, où v est une variable et, a et b sont des valeurs. Posons alors la question de savoir si P satisfait la spécification $[v > 0;\ v = 2]$ dans laquelle le premier prédicat est une précondition et le second prédicat est une postcondition. La réponse est oui si et seulement si $(a > 0 \rightarrow b = 2)$. Autant donc écrire directement $(v > 0 \rightarrow v' = 2)$ au lieu de $[v > 0;\ v = 2]$.

Considérer les spécifications comme des prédicats facilite la composition des spécifications. Pour exprimer le fait qu'un programme doit satisfaire les spécifications S et T, l'expression $(S \wedge T)$ suffit. La composition est moins directe avec les approches qui utilisent les pré et postconditions. Par exemple, pour exprimer le besoin de satisfaire $S := [P(v);\ Q(v)]$ et $T := [V(v);\ W(v)]$ il faut écrire :

$$[v = v_i \wedge (P(v) \vee V(v));\ (P(v_i) \rightarrow Q(v)) \wedge (V(v_i) \rightarrow W(v))]$$

On ne peut donc pas simplement utiliser les définitions S et T, il faut avoir accès à leurs

structure interne pour pouvoir les composer.

Spécifications et Programmes. Selon Hehner un programme est une spécification suffisamment concrète pour être exécutable. Une démarche de spécification met l'accent sur la clarté et la lisibilité alors qu'une démarche de programmation met l'accent sur l'exécutabilité et l'efficacité. Un langage de programmation doit donc permettre de spécifier et de programmer. Ce langage doit être accompagné par une théorie permettant de passer des spécifications aux programmes. Lorsque la spécification la plus claire possible correspond à un programme (par exemple $\mathsf{fact}\ n := 0 \Rightarrow 1 \mid n \Rightarrow n \times (\mathsf{fact}\ (n-1))$), autant utiliser directement ce programme comme spécification. Mais il arrive que la spécification la plus claire possible soit trop abstraite pour être exécutable, c'est la raison pour laquelle le langage de programmation doit inclure des éléments permettant d'écrire des spécifications qui ne sont pas nécessairement exécutables mais sur lesquelles on peut tout de même raisonner.

On peut noter que ceci rejoint la démarche d'Abrial qui généralise les substitutions habituellement utilisées en programmation pour en faire des instruments légitimes de spécification [2]. Dans la même veine la notion de *ghost code* [27] a été introduite dans la plateforme *Why3* [28] afin de pouvoir utiliser des programmes pour spécifier, et ainsi faciliter les spécifications et les preuves. Plus anciennement, le besoin d'intégration entre langage de spécification et langage de programmation avait abouti à la démarche de Morgan consistant à définir une instruction de spécification. Néanmoins, l'approche de Morgan nécessite des règles spécifiques de raisonnement pour ces instructions de spécification. Les approches qui ont hérité de cette tradition n'échappent pas à cette nécessité. Dans l'approche de Hehner, un programme est une spécification, et une spécification est un prédicat, donc un programme est un prédicat. Par conséquent, un programme P satisfait une spécification S (P raffine S) si $P \to S$, les règles de la logique suffisent donc pour raisonner sur les spécifications (et les programmes).

Plus précisément, la traduction d'un programme en prédicat se fait inductivement sur la syntaxe en appliquant les règles suivantes :

- $v := E$ est équivalent à la spécification $v' = E \ \wedge \ w'_1 = w_1 \ \wedge \ ... \ \wedge \ w'_n = w_n$, où les variables w_1 à w_n sont toutes les autres variables qui ne sont pas concernées par l'affectation
- si P et Q sont des programmes, alors

if C **then** P **else** Q est équivalent à $(C \wedge P) \vee (\neg C \wedge Q)$

- si P et Q sont des programmes alors
  P**;**Q est équivalent à $\exists\ v'' \cdot P_{[v''/v']} \wedge Q_{[v''/v]}$
  l'entrée de $P;Q$ est l'entrée v de P, la sortie v' de P est identifiée à l'entrée v de Q en assignant à v' dans P et à v dans Q le même nom v'', et la sortie de $P;Q$ est la sortie v' de Q

La traduction des boucles en prédicats est plus délicate, car cela nécessiterait l'introduction d'un opérateur de point fixe. Mais l'objectif de la traduction en prédicat est in fine de montrer qu'une itération satisfait une spécification. Dans la section ci-dessous consacrée au raffinement, Nous verrons comment établir qu'une itération satisfait une spécification.

L'intégration harmonieuse entre spécifications et programmes fait que les connecteurs habituels de programmation s'appliquent aux spécifications et vice-versa. Par exemple, si S et T sont des spécifications, la spécification qui dit : *se comporter conformément à S, et ensuite se comporter conformément à T* s'exprime aisément avec le connecteur séquentiel '**;**', il suffit d'écrire S**;**T. De même, le connecteur $\vee$ permet d'exprimer le non déterminisme en programmation de manière assez naturelle : $v := v + 1 \ \vee \ w := w - 1$ est équivalent $(v' = v + 1 \wedge w' = w) \ \vee \ (v' = v \wedge w' = w - 1)$. Il s'agit là d'un choix non déterministe entre l'incrémentation de v et la décrémentation de w, et ce choix sera probablement exclusif puisqu'une implémentation intéressante ne peut à la fois satisfaire $(v' = v + 1)$ et $(v' = v)$.

Dans une démarche de développement par raffinements, il est préférable de ne pas découvrir au dernier raffinement que le programme obtenu n'est pas implémentable. Pour cela il est utile de s'assurer que les spécifications sont réalisables à chaque niveau de raffinement. Pour ce faire, on peut utiliser la définition suivante :

une spécification S est *réalisable* ssi $\forall\ v \cdot \exists\ v' \cdot S$

Par exemple, $v' = v/(5 - v)$ n'est pas réalisable car $\forall\ v \cdot \exists\ v' \cdot v' = v/(5 - v)$ n'est pas valide pour $v = 5$.

Raffinement. Dans la démarche prônée par Hehner, une spécification T raffine une spécification S si et seulement si $(T \rightarrow S)$. La relation de raffinement se résume à une simple implication et aucun formalisme supplémentaire n'est nécessaire. En comparaison, dans la démarche de Morgan, $[X;\ Y]$ raffine $[V;\ W]$ si et seulement si $(V \rightarrow X) \wedge (Y \rightarrow W)$,

ce qui est une expression moins directe, sans compter que l'utilisation de variables fantômes peut s'avérer nécessaire dans le cas où la postcondition dépend de l'état initial. De plus, un formalisme spécifique nécessite de nouvelles lois de raisonnement tels que l'affaiblissement des préconditions, ou le renforcement des postconditions. Dans l'approche préconisée par Hehner ceci n'est pas nécessaire car les règles de la logique subsument ces lois spécifiques de raffinement. La méthode de développement par raffinements successifs repose sur la transitivité de la relation de raffinement, cette propriété de la relation de raffinement est une conséquence naturelle de la transitivité de l'implication.

Il y a cependant une difficulté lorsqu'on cherche à montrer que **while** C **do** P **done** raffine S. En effet, il n'est pas aisé de traduire une boucle en prédicat. Néanmoins, pour montrer que **while** C **do** P **done** $\rightarrow S$, il suffit de montrer que **if** C **then** P**;**S **end** $\rightarrow S$. Avec cette règle, la théorie de Hehner est bien entendu incomplète, mais comme la logique de Hoare, elle est relativement complète (c'est-à-dire qu'elle est complète à condition que le langage de spécification soit suffisamment expressif). Car, si **while** C **do** P **done** $\rightarrow S$ est vrai, il existe toujours une spécification R telle que **while** C **do** P **done** $\rightarrow R$ est prouvable et $R \rightarrow S$. la spécification R est analogue à l'invariant de boucle dans la logique de Hoare ou dans TLA [45]. Mais R est plus générale car R n'a pas nécessairement besoin d'être invariant, tout prédicat qui permet de prouver le raffinement est valable comme spécification intermédiaire R.

Pour montrer la correction d'un raffinement, il suffit de traduire le problème en termes logiques. Pour ce faire on pourra appliquer les règles suivantes :

- $v := E \rightarrow S$ est équivalent à $(v' = E \ \wedge \ w'_1 = w_1 \ \wedge \ ... \ \wedge \ w'_n = w_n) \ \rightarrow \ S$
- **if** C **then** P **else** $Q \ \rightarrow \ S$ est équivalent à $(C \wedge P) \vee (\neg C \wedge Q) \ \rightarrow \ S$
- P**;**$Q \ \rightarrow \ S$ est équivalent à $\exists \, v'' \cdot P_{[v''/v']} \wedge Q_{[v''/v]} \ \rightarrow S$
- **while** C **do** P **done** $\rightarrow \ S$ si **if** C **then** P**;**S **end** $\rightarrow \ S$

En notation Z, les spécifications sont aussi des prédicats. La spécification (true) est satisfaite par tous les programmes qui terminent et la spécification (false) est satisfaite par tous les programmes qui ne terminent pas. Cette interprétation est conforme à la théorie des types dans laquelle un programme qui ne termine pas permet de construire une preuve de (false). Mais une conséquence est que, contrairement à la démarche de Hehner, la relation de raffinement ne se résume plus seulement à une implication, mais s'exprime avec plus de complexité comme suit :

$$\forall\ v \cdot (\exists\ v' \cdot S)\ \rightarrow\ ((\exists\ v' \cdot T)\ \wedge\ (\forall\ v' \cdot T \rightarrow S))$$

De plus, la non terminaison peut effectivement être une exigence dans certaines circonstances, par exemple pour assurer une continuité de service. Dans le démarche de Hehner, tous les programmes satisfont (true) car cette expression est valide en toutes circonstances et aucun programme ne satisfait (false) car cette expression est invalide en toutes circonstances. On peut dire l'interprétation de Hehner est plus proche de l'intuition selon laquelle (false) représente le vide et (true) l'espace de tous les programmes.

3.3 Le raffinement en pratique

Pour simplifier l'application des théories exposées précédemment, des règles de raffinement plus simples en ont été dérivées. Ceci a mené au *refinement calculus* de Morgan [57] qui est à la base de la méthode *B-classique* [2], ainsi qu'à celui de Back et von Wright [8]. Le *refinement calculus* naît avec l'instruction de spécification et l'interprétation de cette dernière dans le wp-calcul. L'objectif est alors de pouvoir partir d'une spécification formelle exprimée par une instruction de spécification, puis de raffiner cette spécification en appliquant des règles de raffinement qui garantissent la correction des étapes de raffinement.

La logique de Hoare et le wp-calcul s'appliquent aux programmes exécutables. Or la technique des raffinements successifs peut produire des programmes qui ne sont pas exécutables durant les étapes intermédiaires de raffinements, notamment en raison de la présence de spécifications insuffisamment précises pour pouvoir être compilées. Les instructions de spécification [6, 56] ont été introduites afin que ces spécifications, insuffisamment précises pour pouvoir être compilées, soient tout de même suffisamment formelles pour permettre des preuves rigoureuses de correction, on dispose alors d'un même langage pour spécifier et de programmer à différents niveaux d'abstraction. Une instruction de spécification est notée par : $[P;\ Q]$. Cette instruction spécifie que si le prédicat P est vrai juste avant l'exécution de l'instruction alors le prédicat Q sera vrai immédiatement après l'exécution de l'instruction. Afin de pouvoir raisonner sur ces instructions on leur attribue la définition suivante dans le wp-calcul :

$$\mathsf{wp}\ [P;\ Q]\ R\ \stackrel{\text{def}}{=}\ P \wedge (\forall\ v \cdot Q \rightarrow R)$$

Cependant, le wp-calcul repose en partie sur l'axiome du *miracle exclu* :

$$\forall\ S \cdot (\mathsf{wp}\ S\ \mathsf{False}) = \mathsf{False}$$

Or, l'introduction des instructions de spécification permet potentiellement de violer cet axiome. En effet :

$$(\mathsf{wp}\ [\mathsf{True};\ \mathsf{False}]\ \mathsf{False}) \ = \ \mathsf{True} \wedge (\mathsf{False} \rightarrow \mathsf{False}) \ = \ \mathsf{True}$$

Lorsque l'axiome du *miracle exclu* n'est pas valide cela signifie que l'on est en présence d'une spécification miraculeuse [56], c'est-à-dire une spécification pour laquelle on ne peut générer de code exécutable, la souplesse des instructions de spécification vient avec cette limitation. En général l'objectif est d'aboutir à un programme exécutable, par conséquent il faut s'assurer, durant le processus de raffinement, que l'on n'introduit pas de spécification miraculeuse, et pour cela on pourra utiliser la définition suivante :

$$[P;\ Q] \text{ est réalisable } (\textit{non miraculeuse}) \quad \text{ssi} \quad (\mathsf{wp}\ [P;\ Q]\ \mathsf{False}) = \mathsf{False}$$

Cette définition indique que pour qu'une spécification soit réalisable, il faut qu'il existe au moins un état dans lequel la postcondition Q peut être établie si on suppose que la précondition P est remplie.

Les règles du *refinement calculus* et leur dérivations sont détaillées dans [59]. Nous donnons ci-dessous un bref aperçu de ces règles :

- $(P \rightarrow P') \quad \rightarrow \quad [P';\ Q] \sqsubseteq [P;\ Q]$
- $(Q' \rightarrow Q) \quad \rightarrow \quad [P;\ Q'] \sqsubseteq [P;\ Q]$
- $(P \rightarrow Q) \quad \rightarrow \quad \textbf{skip} \sqsubseteq [P;\ Q]$
- $x := E \quad \sqsubseteq \quad [Q_{[E/x]};\ Q]$
- $([P;\ R]\textbf{;}[R;\ Q]) \quad \sqsubseteq \quad [P;\ Q]$
- $\textbf{if}\ C\ \textbf{then}\ [P \wedge C;\ Q]\ \textbf{else}\ [P \wedge \neg C;\ Q]\ \textbf{end} \quad \sqsubseteq \quad [P;\ Q]$
- $\textbf{while}\ C\ \textbf{do}\ [I \wedge C;\ I]\ \textbf{end} \quad \sqsubseteq \quad [I;\ I \wedge \neg C]$

Ces règles sont plus accessibles en pratique car elles font uniquement intervenir les données du raffinement à prouver, il n'y a pas de quantification sur des prédicats.

Au delà des raffinements individuels, l'historique des raffinements et leur articulation est d'une grande importance car ils permettent de mieux comprendre les développements, et ainsi de réduire le risque d'introduire des erreurs lors des modifications. Afin de formaliser l'articulation des étapes de raffinements, Morgan a introduit dans [58] une

notion de *développement* ainsi qu'un langage dédié. Ce langage permet de représenter un développement sous la forme d'une arborescence des étapes de raffinements qui le constituent. Dans cette arborescence un identifiant est affecté à chaque instruction de spécification, et les raffinements d'une instruction la référencent par son identifiant. Ceci correspond au style de la programmation dite *lettrée* [42] sauf qu'au lieu d'être informelles, les spécifications sont formelles. Les *blocs spécifiés* [32] de Hehner permettent aussi de rendre compte de l'ensemble des raffinements qui constituent un développement. Un bloc spécifié associe une spécification relationnelle et une implémentation. Cette association est syntaxique et les blocs peuvent être imbriqués. Donc ils représentent naturellement une arborescence, et il n'est pas nécessaire de manipuler des identifiants puisque l'ensemble d'un développement est représenté par un arbre syntaxique. Dans [68] Runge et al. proposent une syntaxe permettant de décrire l'arborescence correspondant à une dérivation de programme par raffinements. Les règles de dérivation utilisées sont celles préconisées par Kourie et Watson dans [43]. Ces règles ont leurs équivalents dans le calcul de Morgan mais la théorie sous-jacente est formulée avec la logique de Hoare au lieu du wp-calcul.

3.4 Conclusion

Les approches prédicatives du raffinement fournissent un moyen plus direct pour exprimer les spécifications et les programmes en supprimant un niveau d'indirection par rapport au formalisme relationnel. Parmi les approches prédicatives, la programmation prédicative présente l'avantage d'unifier spécifications et programmes. De plus, la programmation prédicative permet de représenter les programmes comme des relations entre états de programmes. Ceci permet par conséquent de donner une sémantique aux programmes qui est plus simple que celle émanant des approches basées sur la logique de Hoare ou les transformateurs de prédicats. En effet, ces derniers reviennent à interpréter les programmes comme des relations entre ensembles d'états. En pratique, l'approche dominante est celle basée sur les transformateurs de prédicats. Néanmoins, les approches relationnelles ont continué à se développer. En particulier cela a abouti à des points de vue intéressants, notamment à propos de la manière de raisonner sur les boucles. Il serait donc intéressant d'intégrer, dans une théorie prédicative du raffinement les avancées obtenues dans le point de vue relationnel. Pour ce faire, la programmation prédicative, qui peut être vue comme une expression du point de vue relationnel dans le calcul des prédicats, semble très adaptée. De plus, la programmation prédicative représente programmes et spécifications comme des prédicats, ce qui permet de faciliter la mécanisation dans un cadre tel que la théorie des types où les prédicats sont des objets natifs.

Chapitre 4

Langage et Sémantique Prédicative

Dans ce chapitre, nous présentons le langage impératif dans lequel seront exprimés les programmes issus de notre démarche de développement par raffinements. Cette présentation est faite dans le cadre formel de la théorie des types. Étant donné que nous explorons le point de vue relationnel, la première section de ce chapitre introduit une notion générale de spécification qui permet de codifier les relations en théorie des types. La deuxième section décrit le langage de programmation cible dont la sémantique formelle est ensuite présentée dans la troisième section. La quatrième section présente une définition de la notion de raffinement basée sur la sémantique de la section précédente. Dans la dernière section, nous justifions plus formellement notre interprétation des programmes décrite dans la troisième section. D'abord, nous formalisons la notion de raffinement dans le point de vue plus classique de la logique de Hoare. Et ensuite, nous montrons que les deux points de vue sont équivalents au sens où tout raffinement valide dans un point de vue l'est aussi dans l'autre.

4.1 Spécifications et programmes

Le point de vue relationnel consiste à interpréter les programmes et les spécifications uniformément comme des relations mathématiques entre des états initiaux et des états finaux de programmes. Ces relations sont souvent représentées comme des ensembles de paires d'états. En un sens, la programmation prédicative est une expression du point de vue relationnel dans laquelle les relations sont représentées par des prédicats plutôt que dans le formalisme du calcul des relations. Par conséquent, en programmation prédicative, les programmes sont des spécifications, et les spécifications sont des prédicats [31]. Ces prédicats sont dits *avant-après* car ils expriment une relation binaire entre les états d'avant l'exécution du programme et les états consécutifs à l'exécution du programme. Nous

utiliserons la convention habituelle qui consiste à suffixer les noms des variables avec le signe ($'$) pour parler de leurs valeurs après l'exécution du programme.

Exemple 4 (Prédicat avant-après)**.** *Considérons le prédicat suivant :* $x' = x+y \wedge y' = y$*. Ce prédicat désigne un programme dont l'espace des états est constitué de deux variables. Les noms* x *et* y *désignent à la fois les variables ainsi que leurs valeurs initiales avant l'exécution du programme. De manière analogue, les noms* x' *et* y' *désignent les valeurs des variables après l'exécution du programme. Le comportement décrit par le prédicat est le suivant. La valeur* x' *de la variable* x *à la fin de l'exécution est égale à la valeur initiale de* x *augmentée de la valeur initiale de* y*. De plus, la variable* y *conserve sa valeur. Ceci ne veut pas dire que le programme ne modifie pas* y *pendant son exécution. En effet, il est possible que le programme modifie* y*, et qu'ensuite il rétablisse sa valeur à la valeur initiale.*

En théorie des types, les relations entre états de programme peuvent être codifiées aisément puisque les prédicats sont des types, donc des objets natifs de la théorie. Pour représenter les spécifications, on définit comme suit le type des relations binaires entre états de programme.

Définition 10 (Type des relations entre états de programme)**.** *Soient* T *et* T' *deux types génériques. On appelle* $\mathsf{Spec}_{T,T'}$ *le type des relations entre états initiaux de type* T *et états finaux de type* T'*. Si on désigne par* Prop *le type des prédicats alors le type* $\mathsf{Spec}_{T,T'}$ *se définit comme suit :*

$$\mathsf{Spec}_{T,T'} \stackrel{\text{def}}{=} T \to T' \to \mathsf{Prop}$$

Une relation R de type $\mathsf{Spec}_{T,T'}$ est une fonction qui calcule un prédicat à partir d'un état initial e et d'un état final e'. La validité dudit prédicat indique que e' est un résultat acceptable si l'entrée est e. Plus précisément, une telle relation R entre états de programme représente un programme quelconque P tel que :

(1) si P est exécuté dans e, alors P termine si et seulement si $(\exists\ e' \cdot R\ e\ e')$ est vrai

(2) si P termine sur e alors l'état final e' est tel que $R\ e\ e'$ est vrai

Dans le point de vue relationnel, tel que défini par exemple dans [70], on considère qu'un programme ne termine pas sur un état initial e dès lors que ce dernier boucle indéfiniment, diverge, ou fini par arrêter de s'exécuter dans un état erroné. On peut donc considérer que le prédicat R défini, par compréhension, l'ensemble des couples entrée-sortie admissibles d'un programme. En notation ensembliste classique, cet ensemble s'écrirait comme suit:

$$\{\ (e, e')\ |\ R\ e\ e'\ \}$$

Exemple 5. *Pour représenter le prédicat de l'exemple 4 (page 30) en théorie des types, on pourra écrire par exemple :*

$$\lambda\ (x, y)\ (x', y') \Rightarrow x' = x + 1\ \wedge\ y' = y$$

Ici, les états de programme sont de type $(\mathsf{nat} \times \mathsf{nat})$*. L'ensemble des comportements décrits par cette spécification peut s'écrire dans la notation ensembliste comme suit :*

$$\{\ ((x, y), (x', y'))\ |\ x' = x + 1\ \wedge\ y' = y\ \}$$

Par mesure de simplification nous pourrons, par la suite, nous limiter à écrire seulement le corps de la spécification (ex. $x' = x+1\ \wedge\ y' = y$) étant donné que l'entête correspondante (ex. $\lambda\ (x, y)\ (x', y') \Rightarrow ...$) s'en déduit aisément.

Cette interprétation des spécifications nous empêche d'ores et déjà de nous appuyer aveuglément sur le tiers-exclu. En effet, admettre la validité du tiers-exclu conduit immédiatement à admettre la validité de la proposition suivante :

$$\forall\ P \cdot \forall\ e \cdot (\exists\ e' \cdot P\ e\ e') \vee \neg(\exists\ e' \cdot P\ e\ e')$$

Or, une preuve de cette proposition permet de décider l'arrêt d'un programme P quelconque sur une entrée e quelconque, ce qui est impossible. Remarquons que pour un état e donné il peut exister plusieurs états e' tels que $R\ e\ e'$ est vrai, ce qui permet de représenter les comportements non déterministes. On notera cependant qu'il n'est pas possible de représenter ainsi les comportements qui consistent pour un état initial donné, à choisir entre terminer sur un état donné e' ou boucler indéfiniment. En effet, si $(\exists\ e' \cdot P\ e\ e')$ est vrai, alors P termine nécessairement sur e et ne peut donc pas choisir de boucler indéfiniment.

Nous définissons ci-dessous, l'inclusion et l'équivalence de spécifications qui sont les analogues de l'inclusion et l'égalité de relations. Logiquement parlant, l'inclusion de spécifications correspond à une implication généralisée, et l'équivalence de spécifications correspond à une équivalence généralisée.

Définition 11 (Inclusion et équivalence de spécifications)**.** *Soient S_2 et S_1 deux spécifications. L'inclusion et l'équivalence de spécifications sont respectivement définies comme suit :*

$$S_2 \subseteq S_1 \stackrel{\text{def}}{=} \forall\ e\ e' \cdot S_2\ e\ e' \rightarrow S_1\ e\ e'$$
$$S_2 \equiv S_1 \stackrel{\text{def}}{=} \forall\ e\ e' \cdot S_2\ e\ e' \leftrightarrow S_1\ e\ e' \stackrel{\text{def}}{=} (S_2 \subseteq S_1) \wedge (S_1 \subseteq S_2)$$

4.2 Un langage impératif classique

On considère le langage impératif dont la syntaxe est donnée dans la Figure 4.1 ci-dessous.

Statement $S^{T,T'}$ ::=

|	**effect** f	(transformateur d'état avec $f : T \rightarrow T'$)
|	$S_1^{T,U}$ **;** $S_2^{U,T'}$	(séquence)
|	**if** C **then** $S_1^{T,T'}$ **else** $S_2^{T,T'}$ **end**	(alternative avec la condition $C : T \rightarrow \mathsf{Prop}$)
|	$\langle R \rangle$	(sous-programmes avec $R : \mathsf{Spec}_{T,T'}$)

Statement $S^{T,T}$ ::=

|	**while** C **do** $S^{T,T}$ **end**	(itération avec la condition $C : T \rightarrow \mathsf{Prop}$)

Figure 4.1: Langage de programmation cible

La syntaxe du langage est paramétrée par le type T des états initiaux de programme, et par le type T' des états finaux de programme. En général T et T' peuvent être distincts, sauf dans le cas des boucles où ils sont nécessairement identiques. Cette syntaxe s'avèrera parfois particulièrement verbeuse pour représenter certaines constructions dérivées comme l'affectation d'une variable ou la notion de variable locale. Cependant, pour l'heure nous cherchons surtout à nous donner les moyens de représenter de manière formelle et typée les constructions habituelles de la programmation impérative.

Transformation d'état. La construction **effect** f représente une transformation de l'état du programme par une fonction pure $f : T \rightarrow T'$. Cette instruction permet de représenter l'affectation d'une variable. Par exemple, dans le cas où l'espace des états est constituée des variables i et j de type entier naturel, l'affectation $i := i + 1$ est représentée par :

$$(\textbf{effect}\ (\lambda\ (s : \mathsf{Nat} \times \mathsf{Nat}) \Rightarrow \textbf{match}\ s\ \textbf{with}\ (i, j) \Rightarrow (i + 1, j)\ \textbf{end}))$$

Autrement dit, le transformateur décrit ci-dessus agit sur un état formé par le couple d'entiers (i, j), et produit un nouvel état $(i + 1, j)$ dans lequel la variable i a été modifiée et la variable j demeure inchangée.

Séquence. La composition séquentielle de deux instructions S_1 et S_2 est définie uniquement lorsque le type U des états finaux de S_1 coïncide avec le type des états initiaux de S_2. Combiné avec la notion générique de transformation d'état, cette possibilité de changement d'état pendant les séquences permet en particulier de représenter les variables locales comme nous le verrons plus loin.

Alternative. L'instruction if-then-else représente le choix alternatif entre deux instructions S_1 et S_2 en fonction de la condition C. Les deux instructions doivent avoir le même type d'état initial et le même type d'état final. La condition C est représentée par une fonction $C : T \to \mathsf{Prop}$ qui calcule une proposition à partir d'un état initial. Par exemple, pour tester si la variable i est supérieure à j, on peut écrire

$$\textbf{if } (\lambda\ (i,j) \Rightarrow i > j) \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}$$

Sous-programmes. L'instruction $\langle R \rangle$ représente un sous-programme dont la relation entre l'état initial et l'état final est définie par le prédicat R dont le type $\mathsf{Spec}_{T,T'}$ est défini ci-dessous. Par exemple pour représenter un sous-programme qui augmente la valeur de la variable i et laisse inchangées les variables j_1 à j_n, on peut écrire :

$$\langle\ \lambda\ (i, j_1, ..., j_n)\ (i', j_1', ..., j_n') : \mathsf{nat})\ \Rightarrow\ i' > i\ \wedge\ j_1' = j_1\ \wedge\ ...\ \wedge\ j_n' = j_n\ \rangle$$

Cependant, pour plus de concision, nous utiliserons la notation suivante : $\langle\ i' > i\ \rangle_{j_1,...,j_n}$

Itération. La condition C de la boucle est analogue à celle de l'alternative. Ici, le corps S de la boucle doit avoir le même type pour ses états initiaux et finaux afin que la répétition de S en séquence (S;S;...;S) soit définie.

4.3 Sémantique prédicative

À chaque programme nous associerons une relation entre états de programme, soit un objet de type $\mathsf{Spec}_{T,T'}$. On définit par conséquent la fonction sémantique $[\![\cdot]\!]$ de type $\mathsf{Statement}^{T,T'} \to \mathsf{Spec}_{T,T'}$, construite par induction sur la syntaxe comme indiqué dans la figure 4.2 ci-dessous. À l'exception des boucles, cette interprétation correspond à une traduction en théorie des types de la sémantique prédicative formulée par Sekerinski dans [70]. Avant d'expliquer plus en détails la sémantique prédicative, nous définissons ci-dessous une notion d'équivalence de programmes qui nous sera utile dans la suite.

$$
\begin{array}{ll}
[\![\textbf{effect } f]\!] & \stackrel{\text{def}}{=} \lambda\ e\ e' \Rightarrow e' = (f\ e) \\
[\![\langle R \rangle]\!] & \stackrel{\text{def}}{=} R \\
[\![S_1;S_2]\!] & \stackrel{\text{def}}{=} [\![S_1]\!] \square [\![S_2]\!] \\
[\![\textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}]\!] & \stackrel{\text{def}}{=} [\![S_1]\!] \lhd C \rhd [\![S_2]\!] \\
[\![\textbf{while } C \textbf{ do } S \textbf{ end}]\!] & \stackrel{\text{def}}{=} (\mathsf{lfp}\ \mathcal{W}_S^C), \quad \text{où } \mathcal{W}_S^C \stackrel{\text{def}}{=} \lambda\ \mathcal{X} \Rightarrow ([\![S]\!] \square \mathcal{X}) \lhd C \rhd [\![\textbf{skip}]\!]
\end{array}
$$

Figure 4.2: Sémantique prédicative

Définition 12. *Soient deux programmes P_1 et P_2, nous dirons P_1 et P_2 sont équivalents si et seulement si leurs sémantiques sont équivalentes.*

$$P_1 \ \equiv \ P_2 \ \stackrel{\text{def}}{=} \ [\![P_1]\!] \ \equiv \ [\![P_2]\!]$$

Détaillons à présent la sémantique prédicative associée à chaque construction de notre langage.

Transformation d'état. La spécification associée à l'instruction (**effect** f) traduit le fait que cette instruction termine sur tous les états d'entrée e puisque f est une fonction totale. Conformément à l'interprétation relationnelle, on peut aisément vérifier que $(\exists\ e' \cdot\ [\![\textbf{effect } f]\!]\ e\ e')$ est une tautologie. De plus, les comportements admissibles de l'instruction (**effect** f) correspondent aux états initiaux e et aux états finaux e' tels que $e' = (f\ e)$, ce qui correspond exactement à $([\![\textbf{effect } f]\!]\ e\ e')$.

Sous-programmes. La spécification R dans $\langle R \rangle$ est déjà un objet de type $\mathsf{Spec}_{T,T'}$. On considérera que le sous-programme $\langle R \rangle$ est directement spécifié par R dans l'interprétation relationnelle expliquée plus haut. La fonction $[\![\cdot]\!]$ associe donc directement R à $\langle R \rangle$.

Alternative. L'instruction if-then-else dénote la spécification décrite ci-dessous par la relation (4.1). Ici, C est de type $T \to \mathsf{Prop}$.

$$[\![S_1]\!] \lhd C \rhd [\![S_2]\!] \stackrel{\text{def}}{=} \lambda\ (e\ e' : T) \Rightarrow (C\ e \wedge ([\![S_1]\!]\ e\ e')) \ \vee \ (\neg(C\ e) \wedge ([\![S_2]\!]\ e\ e')) \qquad (4.1)$$

La notation utilisée est tirée de [34], et exprime un choix entre deux spécifications en fonction de la validité de $(C\ e)$. C'est-à-dire, soit C est vraie ($\lhd$) de l'état initial e et $[\![S_1]\!]$ est choisie, soit C est fausse ($\rhd$) de e et $[\![S_2]\!]$ est choisie. Il est facile de vérifier que l'instruction if-then-else termine sur état initial e si et seulement si $(C\ e)$ est vrai et S_1

termine sur e ou $(C\ e)$ est faux et S_2 termine sur e.

Pour un état initial donné e, la conditions $(C\ e)$ est de type Prop, ce qui signifie que $(C\ e)$ n'est pas nécessairement décidable. Autrement dit, l'assertion $(\forall\ e \cdot (C\ e) \vee \neg(C\ e))$ n'est pas nécessairement prouvable. En conséquence la spécification définie par la relation 4.1 n'est pas équivalente à la formulation ci-dessous souvent utilisée dans les cadres où on considère que la condition $(C\ e)$ est réputée décidable.

$$\lambda\ (e\ e' : T) \Rightarrow (C\ e) \rightarrow (\llbracket S_1 \rrbracket\ e\ e') \quad \wedge \quad \neg(C\ e) \rightarrow (\llbracket S_2 \rrbracket\ e\ e') \tag{4.2}$$

Cependant, si $(C\ e)$ n'est pas décidable alors $(\llbracket S_1 \rrbracket \lhd C \rhd \llbracket S_2 \rrbracket)$ n'est pas habité car ni $(C\ e)$, ni $\neg(C\ e)$ ne sont habités. Autrement dit, $(\llbracket S_1 \rrbracket \lhd C \rhd \llbracket S_2 \rrbracket)$ reflète bien un programme qui ne termine pas sur e lorsque $(C\ e)$ n'est pas décidable, et il est heureux qu'il en soit ainsi puisque l'instruction if-then-else ne peut être exécutée sur e si $(C\ e)$ est indécidable. Si $(C\ e)$ avait été plutôt de type bool alors la décidabilité de la condition aurait été garantie par typage. Néanmoins, cela rendrait l'écriture de certaines spécifications moins commode. Par exemple, $(C\ e)$ et $(\llbracket S_1 \rrbracket\ e\ e')$ n'auraient plus le même type ce qui nuirait à l'uniformité de la définition puisqu'il faudrait alors écrire:

$$\llbracket S_1 \rrbracket \lhd C \rhd \llbracket S_2 \rrbracket \stackrel{\text{def}}{=} \lambda\ (e\ e' : T) \Rightarrow (C\ e) = \mathsf{true} \wedge (\llbracket S_1 \rrbracket\ e\ e') \vee (C\ e) = \mathsf{false} \wedge (\llbracket S_2 \rrbracket\ e\ e') \tag{4.3}$$

De plus à la différence de Prop, le type bool n'est pas clos par quantification universelle ou existentielle. Par exemple, si $(P\ e\ e')$ est de type bool, alors il n'est pas possible de quantifier directement sur e' en écrivant $(\exists\ e' \cdot P\ e\ e')$. Le choix de Prop pour $(\llbracket S \rrbracket\ e\ e')$ facilite donc l'écriture des spécifications, raison de plus pour uniformiser en faveur de Prop.

Séquence. La sémantique de l'opérateur (**;**) utilise une notion particulière de composition séquentielle de spécifications définie ci-dessous.

Définition 13 (Composition de spécifications)**.** *Soient S_1 et S_2 deux objets de types respectifs $\mathsf{Spec}_{T,U}$ et $\mathsf{Spec}_{U,V}$. On définit respectivement la composition angélique et la composition démoniaque [11, 9] comme suit :*

$$\begin{array}{lll} S_1 \boxdot S_2 & \stackrel{\text{def}}{=} & \lambda\ e\ e' \Rightarrow\ \exists\ e_x \cdot S_1\ e\ e_x\ \wedge\ S_2\ e_x\ e' \\ S_1 \square S_2 & \stackrel{\text{def}}{=} & \lambda\ e\ e' \Rightarrow\ (S_1 \boxdot S_2)\ e\ e'\ \wedge\ \forall\ e_x \cdot S_1\ e\ e_x\ \rightarrow\ \exists\ e' \cdot S_2\ e_x\ e' \end{array}$$

La composition angélique est la composition relationnelle classique. Comme l'illustre l'exemple ci-dessous, $S_1 \boxdot S_2$ contient l'état e dans son domaine dès lors qu'il existe un

état e' tel que S_1 associe e à e' et e' est dans le domaine de S_2.

$$\{(1,2),(1,3),(2,4)\} \boxdot \{(2,4),(4,5)\} = \{(1,4),(2,5)\}$$

Cette forme de composition est qualifiée d'angélique en raison de son caractère optimiste qui consiste à considérer que l'existence d'une possibilité de terminaison sur un état initial e donné est suffisante pour accepter la présence de e dans le domaine de la composition. Cependant, ceci n'est pas conforme à l'interprétation relationnelle dans laquelle la présence de e dans le domaine vaut garantie de terminaison. Pour se conformer à l'interprétation relationnelle, nous devons considérer que toute possibilité de non terminaison vaut non terminaison. C'est-à-dire que la présence d'un état e dans le domaine de $\langle S_1 \rangle ; \langle S_2 \rangle$ est conditionné au fait que e' soit présent dans le domaine de S_2 à chaque fois que S_1 associe e' à e. Cette forme de composition est qualifiée de démoniaque en raison de son caractère plus conservateur. La composition démoniaque ajoute donc une restriction à la composition angélique qui la rend compatible avec l'interprétation relationnelle des programmes. Comme le montre l'exemple ci-dessous, l'état (1) est absent du domaine de la composition démoniaque car dans le cas où l'état final après l'exécution de S_1 est (3), la composition échoue puisque l'état (3) n'est pas dans le domaine de S_2.

$$\{(1,2),(1,3),(2,4)\} \Box \{(2,4),(4,5)\} = \{(2,5)\}$$

Notons que l'opérateur $\Box$ est monotone à droite vis-à-vis de l'inclusion des spécifications comme indiqué formellement dans le lemme suivant. Cette propriété nous sera utile pour définir la sémantique des boucles.

Lemme 5 ($\Box$ est monotone à droite)**.** *Soient les spécifications S, X et Y. Alors, on a:*

$$(X \subseteq Y) \rightarrow ((S \Box X) \subseteq (S \Box Y))$$

Preuve. Immédiat après expansion des définitions. □

De même que dans [30], nous interprétons donc la composition séquentielle de deux instructions comme la composition démoniaque de leurs interprétations respectives. La notation $\Box$ est tirée de [30]. Nous utilisons une notation similaire pour la composition angélique afin d'éviter toute confusion avec la composition séquentielle des instructions (;).

Itération. L'instruction while est interprétée comme le plus petit point fixe (noté lfp) de la fonction $\mathcal{W}_P^{\mathcal{C}}$. L'opérateur de point fixe lfp peut être défini en théorie des types par la

fonction suivante:

$$\mathsf{lfp}\ (F : \mathsf{Spec}_{T,T} \to \mathsf{Spec}_{T,T}) \quad \stackrel{\text{def}}{=} \quad \lambda\ e\ e' \Rightarrow \forall\ X \cdot (F\ X \quad \subseteq \quad X) \to X\ e\ e'$$

Le prédicat ($\mathsf{lfp}\ F\ e\ e'$) est vrai si et seulement si ($X\ e\ e'$) est vrai pour toute spécification X telle que la relation symbolisée ($F\ X$) est incluse dans la relation symbolisée par X. Cette définition est justifiée par le fait que la composition démoniaque de spécifications est monotone à droite vis-à-vis de l'ordre d'inclusion des spécifications (lemme 5). Par conséquent, $\mathcal{W}_P^C$ est monotone. Donc, par le théorème du point fixe de Knaster-Tarski [74], ($\mathsf{lfp}\ \mathcal{W}_P^C$) est défini puisque les relations binaires ordonnées par l'inclusion ensembliste forment un treillis complet. L'opérateur lfp n'est qu'une traduction en théorie des types de la caractérisation ensembliste du plus petit point fixe donnée par le théorème de Knaster-Tarski. On a par conséquent le lemme suivant.

Lemme 6. *Soit C une condition, et soit P un programme. On a les propriétés suivantes :*

(1) **while** C **do** P **end** $\equiv$ **if** C **then** P;(**while** C **do** P **end**) **end**
(2) $\forall\ X \cdot (\llbracket$ **if** C **then** $P;\langle X \rangle$ **end** $\rrbracket \subseteq X) \to (\llbracket$**while** C **do** P **end**$\rrbracket \subseteq X)$

Preuve. Conséquence immédiate du théorème du point fixe de Knaster-Tarski. □

La propriété (1) atteste que la sémantique de la boucle est bien un point fixe de $\mathcal{W}_P^C$. Cette propriété permet en particulier de raisonner par cas en simulant, à un niveau logique, l'exécution d'un tour de boucle. La propriété (2) est le principe d'induction qui découle du fait que la sémantique de la boucle est le plus petit point fixe de $\mathcal{W}_P^C$. Ce principe d'induction est utile pour prouver des assertions de la forme ($\llbracket$**while** C **do** P **end**$\rrbracket \subseteq X$), où X est une spécification.

Si la boucle termine sur l'état initial e, alors tous les chemins d'exécution à partir de e sont finis et aboutissent à un état e' tel que $\neg(C\ e')$ est vrai. Étant donné que le non-déterminisme n'est pas borné, il peut exister une infinité de chemins (finis) d'exécutions à partir de e. Si ($\exists\ e' \cdot \llbracket$**while** C **do** P **end**$\rrbracket\ e\ e'$) est vrai, le théorème 1 ci-dessous permet de justifier que la notion de point fixe choisie correspond bien à l'interprétation relationnelle d'une boucle while, c'est-à-dire que l'exécution de la boucle à partir de l'état initial e termine nécessairement. La preuve de ce théorème s'appuie sur le lemme suivant.

Lemme 7. *Soit C une condition, et soit P un programme, alors on a:*

$$\begin{array}{l}\forall\ Y \cdot (\exists\ e' \cdot [\![\textsf{while}\ C\ \textsf{do}\ P\ \textsf{end}]\!]\ e\ e') \\ \qquad \rightarrow (\forall\ e \cdot ((C\ e \wedge (\forall\ e_x \cdot P\ e\ e_x \rightarrow Y\ e_x)) \vee \neg(C\ e)) \rightarrow Y\ e) \rightarrow Y\ e\end{array}$$

Schéma de preuve. Par induction sur la sémantique de la boucle en appliquant la propriété (2) du lemme 6. Pour le paramètre X de cette propriété, on choisira $(\lambda\ e\ e' \Rightarrow Y\ e)$. □

Théorème 1. *La relation $\prec_P^C$ définie ci-après est une relation bien fondée.*

$$\prec_P^C \ \stackrel{\text{def}}{=}\ \lambda\ e\ e'\ \Rightarrow\ C\ e'\ \wedge\ [\![P]\!]\ e'\ e\ \wedge\ (\exists\ e'' \cdot\ [\![\textsf{while}\ C\ \textsf{do}\ P\ \textsf{end}]\!]\ e'\ e'')$$

Schéma de preuve. On suppose qu'on a un état e' dans le domaine de $[\![\textsf{while}\ C\ \textsf{do}\ P\ \textsf{end}]\!]$. Puis on applique le lemme 7 pour montrer que pour tout e, si $(C\ e'\ \wedge\ [\![P]\!]\ e'\ e\ )$ est vrai alors e est accessible par $\prec_P^C$. Ayant ainsi montré que tous les prédécesseurs de e' par $\prec_P^C$ sont accessibles, nous avons que e' est accessible par $\prec_P^C$. □

La relation $\prec_P^C$ est une restriction de la relation $(\lambda\ e\ e'\ \Rightarrow\ C\ e'\ \wedge\ [\![P]\!]\ e'\ e)$ au domaine de $[\![\textsf{while}\ C\ \textsf{do}\ P\ \textsf{end}]\!]$. Le théorème 1 montre qu'il ne peut pas exister de chaîne infinie d'exécution de la boucle commençant par l'état initial e. En effet, supposons qu'il existe une exécution infinie de la boucle à partir d'un état e, alors on a la chaîne infinie de la forme suivante:

$$(\exists\ e' \cdot [\![\textsf{while}\ C\ \textsf{do}\ P\ \textsf{end}]\!]\ e\ e')\ \ \wedge\ \ C\ e \wedge [\![P]\!]\ e\ e_1\ \wedge\ C\ e_1 \wedge [\![P]\!]\ e_1\ e_2\ \wedge\ ...$$

Il suffit d'utiliser la propriété (1) du lemme 6 pour voir que la chaîne ci dessus est équivalente à la chaîne infinie descendante qui suit:

$$...\ \ e_3\ \ \prec_p^C\ \ e_2\ \ \prec_p^C\ \ e_1\ \ \prec_p^C\ \ e$$

Comme $\prec_P^C$ est une relation bien fondée, une telle chaîne qui correspondrait à une boucle infinie est impossible lorsque e est dans le domaine de $[\![\textsf{while}\ C\ \textsf{do}\ P\ \textsf{end}]\!]$. Donc, par contradiction et conformément à l'interprétation relationnelle des programmes, on a bien que la boucle termine sur l'état initial e si et seulement si $(\exists\ e' \cdot [\![\textsf{while}\ C\ \textsf{do}\ P\ \textsf{end}]\!]\ e\ e')$ est vrai.

On notera que, de manière analogue à l'instruction if-then-else, le fait que la condition C n'est pas nécessairement décidable est pris en compte dans la sémantique de l'instruction while. En effet, le type $([\![\textsf{while}\ C\ \textsf{do}\ P\ \textsf{end}]\!]\ e\ e')$ n'est pas habité lorsque ni $(C\ e)$, ni $\neg(C\ e)$ ne le sont.

4.4 Constructions dérivées

Le langage que nous avons défini permet de représenter indirectement d'autres constructions habituelles des langages impératifs comme par exemple les variables locales. Nous montrons rapidement comment émuler ces constructions. On notera en particulier qu'il est envisageable d'étendre plus tard la syntaxe pour permettre une représentation explicite des constructions en question sans pour autant remettre en cause la sémantique que nous venons d'expliquer.

Élément neutre de (;) L'instruction **skip** correspond à un programme n'ayant aucun effet. Donc, nous représenterons **skip** comme une affectation dans laquelle le paramètre f est la fonction identité, soit :

$$\textbf{skip} \stackrel{\text{def}}{=} \textbf{effect}\ (\lambda\ (e : T) \Rightarrow e)$$

On peut aisément vérifier que **skip** possède la propriété suivante :

$$\forall\ P \cdot\ \ \textbf{skip;}P \equiv P\textbf{;skip} \equiv P$$

Élément absorbant de (;) L'instruction **abort** qui correspond à un programme qui ne termine jamais est formalisée comme un cas particulier de sous-programme avec un ensemble vide de comportements acceptables, c'est-à-dire que **abort** est définie par le sous-programme dont la spécification est le prédicat False:

$$\textbf{abort} \stackrel{\text{def}}{=} \langle\ \lambda\ e\ e' \Rightarrow \mathsf{False}\ \rangle$$

On vérifie aisément que **abort** possède la propriété suivante :

$$\forall\ P \cdot\ \ \textbf{abort;}P \equiv P\textbf{;abort} \equiv \textbf{abort}$$

Variables locales. Comme indiqué dans [30], la déclaration d'une variable locale peut être vue comme une transformation d'état. En effet, partant d'un état e de type T, déclarer et initialiser une variable locale v de type U revient à construire un état e' de type $T' = T \times U$ contenant en plus de e la valeur de v. Ceci peut être représentée avec l'instruction **effect** comme suit :

$$\textbf{effect}\ (\lambda\ e \Rightarrow (e, v))$$

De manière analogue, la sortie de v de l'environnement lexical correspond à une projection d'un état $e' : T \times U$ vers un état $e : T$ sans la valeur de la variable v. Ceci peut également être représentée avec l'instruction **effect** comme suit :

$$\textbf{effect}\ (\lambda\ (e, v) \Rightarrow e)$$

Le cycle de vie d'une variable locale est définie par la séquence habituelle : entrée dans l'environnement lexical, utilisation de la variable, et sortie de l'environnement lexical. Par exemple :

$$\begin{array}{lcl} \textbf{var}\ t := x\ \textbf{in} & & (\textbf{effect}\ (\lambda\ (x, y) \Rightarrow (x, y, x))\textbf{;} \\ \quad x := y\textbf{;}\ y := t & \stackrel{\text{def}}{=} & (\textbf{effect}\ (\lambda\ (x, y, t) \Rightarrow (y, y, t))\textbf{;}\ (\textbf{effect}\ (\lambda\ (x, y, t) \Rightarrow (x, t, t))\textbf{;} \\ \textbf{end} & & (\textbf{effect}\ (\lambda\ (x, y, \_) \Rightarrow (x, y)) \end{array}$$

4.5 Raffinement relationnel de programmes

La relation de raffinement, notée $(S_2 \sqsubseteq S_1)$, lie deux programmes S_2 et S_1 tels que S_2 peut prendre la place de S_1 tout en offrant a minima les mêmes fonctionnalités aux utilisateurs (humains ou non). Autrement dit, toute spécification satisfaite par S_1 est aussi satisfaite par S_2. En général S_2 est une version améliorée de S_1. Par exemple S_1 pourrait être une simple spécification non exécutable du comportement désiré d'un programme, alors que S_2 pourrait être un programme exécutable qui dans ce cas remplirait les fonctions attendues et spécifiées par S_1. Autre exemple, S_2 pourrait être un programme qui remplit les mêmes fonctions que S_1 tout en étant plus performant en temps ou en espace. Avant de discuter plus amplement les propriétés de la relation de raffinement, nous donnons d'abord ci-dessous une définition formelle de $(\sqsubseteq)$. Cette définition est une traduction en théorie des types de la définition utilisée dans le formalisme de la notation Z [73].

Définition 14 (Raffinement)**.** *Soient S_2 et S_1 deux programmes, on dit que S_2 raffine S_1 si et seulement si à chaque fois que S_1 termine sur un état initial e, S_2 aussi termine sur e, et tous les résultats acceptables de S_2 pour l'entrée e sont aussi des résultats acceptables de S_1 pour e:*

$$S_2 \sqsubseteq S_1 \stackrel{\text{def}}{=} \forall\ e \cdot ((\exists\ e' \cdot [\![S_1]\!]\ e\ e') \rightarrow (\forall\ e' \cdot [\![S_2]\!]\ e\ e' \rightarrow [\![S_1]\!]\ e\ e') \wedge (\exists\ e' \cdot [\![S_2]\!]\ e\ e'))$$

Cette définition peut aussi se comprendre de la manière suivante. On considère que S_2 raffine S_1 si et seulement si on a à la fois :

(1) une réduction du non déterminisme : $[\![S_2]\!]$ est moins non-déterministe que $[\![S_1]\!]$
(2) un élargissement du domaine : le domaine de $[\![S_2]\!]$ contient au moins celui de $[\![S_1]\!]$

Exemple 6 (Exemples de raffinements)**.**

(1) $\forall\ P \cdot P \sqsubseteq \textbf{abort}$*, puisque* $\forall\ e\ e' \cdot [\![\textbf{abort}]\!]\ e\ e' \ \leftrightarrow\ \mathit{False}$
(2) $\{(e, f_0)\} \sqsubseteq \{(e, f_0), (e, f_1)\}$ *par réduction du non-déterminisme*
(3) $\{(e, f_0), (g, h)\} \sqsubseteq \{(e, f_0)\}$ *par élargissement du domaine*
(4) $\{(e, f_0), (g, h)\} \sqsubseteq \{(e, f_0), (e, f_1)\}$ *par réduction et par élargissement*
(5) $i := i + 1 \sqsubseteq \langle i < i' \rangle$ *par réduction du non-déterminisme*

Comme l'indique le lemme 8 ci-dessous, la relation de raffinement est un ordre partiel. En fait l'ensemble des programmes ordonnés par $(\sqsubseteq)$ forment un semi-treillis dont le plus grand élément est **abort**. Il n'existe pas de programme qui raffinerait toutes les autres, donc il n'existe pas de plus petit élément par $(\sqsubseteq)$.

Lemme 8 ($\sqsubseteq$ est un ordre partiel)**.** *Soient les programmes P, Q et R. On a les propriétés suivantes :*

$$\begin{array}{lll} (1) & P \sqsubseteq P & \\ (2) & P \sqsubseteq Q \ \rightarrow\ Q \sqsubseteq P & \rightarrow\ \ P \equiv Q \\ (3) & P \sqsubseteq Q \ \rightarrow\ Q \sqsubseteq R & \rightarrow\ \ P \sqsubseteq R \end{array}$$

Preuve. Immédiate à partir de la définition de $\sqsubseteq$. □

La propriété de transitivité permet de procéder par raffinements successifs pour aller d'une spécification S à un programme exécutable P. En effet, toutes les étapes intermédiaires de raffinements satisfont par transitivité la spécification initiale S:

$$P \sqsubseteq R_n \sqsubseteq R_{n-1}\ ...\ \sqsubseteq R_1 \sqsubseteq R_0 \sqsubseteq S \ \ \rightarrow\ \ P \sqsubseteq S$$

La relation de raffinement a aussi la propriété d'être monotone vis-à-vis des structures de contrôle, ceci est l'objet du lemme 9 ci-dessous. Cette propriété est importante car elle permet de raffiner par parties. Par exemple, pour raffiner $P_1;P_2$, on raffine d'abord P_1 en construisant Q_1, on raffine ensuite P_2 en construisant Q_2, et enfin on combine ces deux raffinements pour obtenir $Q_1;Q_2$. On a alors que, par construction, $Q_1;Q_2$ est un raffinement de $P_1;P_2$. L'intérêt du raffinement par parties réside dans le fait (ou l'espoir) que le raffinement de chacune des parties est potentiellement plus simple que le raffinement direct du tout.

Lemme 9 (Monotonie vis-à-vis de $\sqsubseteq$)**.** *Désignons par les lettres P et Q des programmes, et soit C une condition. On a les propriétés suivantes:*

(1) $P_1 \sqsubseteq Q_1 \;\rightarrow\; P_2 \sqsubseteq Q_2 \;\rightarrow\; \textsf{if } C \textsf{ then } P_1 \textsf{ else } P_2 \textsf{ end} \;\sqsubseteq\; \textsf{if } C \textsf{ then } Q_1 \textsf{ else } Q_2 \textsf{ end}$

(2) $P_1 \sqsubseteq Q_1 \;\rightarrow\; P_2 \sqsubseteq Q_2 \;\rightarrow\; P_1;P_2 \;\sqsubseteq\; Q_1;Q_2$

(3) $P \sqsubseteq Q \;\rightarrow\; \textsf{while } C \textsf{ do } P \textsf{ end} \;\sqsubseteq\; \textsf{while } C \textsf{ do } Q \textsf{ end}$

Schéma de preuve. Les propriétés (1) et (2) se prouvent aisément à partir des définitions. La propriété (2) en particulier utilise le fait que l'opérateur ($\square$) est monotone vis-à-vis de ($\sqsubseteq$). Considérons maintenant la propriété (3), et supposons que $P \sqsubseteq Q$. Il reste à montrer que **while** C **do** P **end** $\sqsubseteq$ **while** C **do** Q **end**. On montre d'abord la réduction du non-déterminisme par induction sur **while** C **do** P **end** en appliquant la propriété (2) du lemme 6. Ensuite, on montre l'élargissement du domaine par induction bien fondée sur $(\prec_Q^C)$. D'où la validité du raffinement. □

4.6 Raffinement de programmes en logique de Hoare

Nous avons présenté une sémantique prédicative formalisée en théorie des types que nous avons justifié avec des arguments informels. Nous allons maintenant formaliser, toujours en théorie des types, un autre point de vue sur la notion de raffinement basée sur une logique de programmes plus classique à la Hoare [35]. Nous montrerons ensuite l'équivalence des deux points de vue, espérant avoir ainsi justifié plus formellement la pertinence de notre interprétation prédicative des programmes en théorie des types.

4.6.1 Logique de Hoare

En logique de Hoare, les spécifications sont des paires de conditions (P, Q). La condition P est appelée précondition et définit l'ensemble des états initiaux dans lequel un programme est supposé démarrer son exécution. La condition Q est appelée postcondition et définit l'ensemble des états finaux dans lesquels un programme termine son exécution. Le fait qu'un programme S satisfait une spécification (P, Q) est notée par le prédicat $[P]\ S\ [Q]$ appelé triplet de Hoare.

Définition 15 (Triplet de Hoare)**.** *Le triplet $[P]\ S\ [Q]$ est un prédicat valide si et seulement si pour tout état initial e tel que $(P\ e)$ est vrai, S termine et l'état final e' est tel que $(Q\ e')$ est vrai.*

4.6.1.1 Le système de preuves

La famille des triplets de Hoare valides est spécifiée par les règles de constructions de la figure 4.3 ci-dessous. Dans [17], ces règles ont été prouvées absolument correctes, et complètes relativement à l'expressivité du langage permettant de formaliser les spécifications. Cependant, il convient de préciser que le cadre formel qui sous-tend cette preuve est différent de celui de la théories des types. Nous allons maintenant expliciter brièvement les règles de la logique qui sont globalement standards mais avec quelques adaptations liées à la codification en théorie des types.

$$\frac{\forall\ e \cdot P\ e\ \rightarrow\ Q\ (f\ e)}{[P]\ (\textbf{effect}\ f)\ [Q]}\ (\text{effet}) \qquad \frac{[P]\ S_1\ [X] \quad [X]\ S_2\ [Q]}{[P]\ (S_1\ ;\ S_2)\ [Q]}\ (\text{seq})$$

$$\frac{\begin{array}{l}\forall\ e \cdot\ P\ e \rightarrow \mathsf{decidable}\ (C\ e)\\ [\ \lambda\ e \Rightarrow\ C\ e\ \wedge\ P\ e\ ]\ S_1\ [Q]\\ [\ \lambda\ e \Rightarrow\ \neg C\ e\ \wedge\ P\ e\ ]\ S_2\ [Q]\end{array}}{[P]\ (\textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end})\ [Q]}\ (\text{alternative})$$

$$\frac{\begin{array}{l}\mathsf{wfd}\ (\prec)\\ \forall\ e \cdot\ I\ e \rightarrow \mathsf{decidable}\ (C\ e)\\ \forall\ v \cdot\ [\ \lambda\ e \Rightarrow\ C\ e\ \wedge\ I\ e\ \wedge\ v = (V\ e)\ ]\ S\ [\ \lambda\ e' \Rightarrow\ I\ e'\ \wedge\ (V\ e') \prec v\ ]\\ \forall\ e \cdot\ P\ e\ \rightarrow\ I\ e\\ \forall\ e' \cdot\ \neg C\ e'\ \wedge\ I\ e'\ \rightarrow\ Q\ e'\end{array}}{[P]\ (\textbf{while}\ C\ \textbf{do}\ S\ \textbf{end})\ [Q]}\ (\text{itération})$$

$$\frac{\forall\ e \cdot P\ e \rightarrow\ (\exists\ e' \cdot R\ e\ e')\ \wedge\ (\forall\ e' \cdot R\ e\ e' \rightarrow Q\ e')}{[P]\ \langle R \rangle\ [Q]}\ (\text{sub})$$

Figure 4.3: Une logique de Hoare codifiée en théorie des types

Transformation d'état. Par définition, le programme (**effect** f) termine systématiquement dans l'état $(f\ e)$. Par conséquent, il suffit que $(Q\ (f\ e))$ soit vrai à chaque fois que $(P\ e)$ est vrai, pour que $\{P\}$ (**effect** f) $\{Q\}$ soit valide.

Séquence. La règle de la séquence se comprend comme suit. Soit X une condition donnée. Pour que le triplet $\{P\}\ S_1;S_2\ \{Q\}$ soit valide il faut réunir les garanties suivantes. D'une part, si S_1 s'exécute à partir d'un état dans P, il faut que S_1 termine, et que tous ses états finaux soient dans X. Ceci est l'objet de la première prémisse. D'autre part, tous les états dans X doivent être des états initiaux qui permettent à S_2 de terminer, et

de plus tous les états finaux de S_2 doivent tomber dans Q. Ceci est l'objet de la seconde prémisse.

Alternative. L'instruction if-then-else ne peut être exécutée à moins que la condition C ne soit décidable. Par conséquent, pour que cette instruction termine sur un état initial e tel que $(P\ e)$ est vrai, il est nécessaire que $(C\ e)$ soit décidable, d'où la première prémisse de la règle correspondante. Supposons que l'exécution démarre sur une état e tel que $(P\ e)$ est vrai. Dans le cas où $(C\ e)$ est vrai, il suffit que S_1 termine sur un état e' tel que $(Q\ e')$ est vrai pour que l'instruction if-then-else en fasse de même. Ceci revient à associer à S_1 la précondition $(\lambda\ e \Rightarrow C\ e\ \wedge\ P\ e)$, d'où le deuxième prémisse, ainsi que la troisième de manière analogue.

Itération. La règle d'itération fait intervenir les concepts d'invariant et de variant (notés I et V). De plus, comme le symbolise la première prémisse, l'exigence de terminaison nécessite un ordre bien fondé que nous noterons $\prec$. On suppose que la boucle s'exécute à partir d'un état initial qui satisfait l'invariant I. Par conséquent, pour les mêmes raisons que dans le cas de l'alternative, il est nécessaire que $(C\ e)$ soit décidable pour tout état e tel que $(I\ e)$ est vrai. C'est justement ce que permet de garantir la deuxième prémisse.

La troisième prémisse a deux objectifs. Le premier est de garantir la terminaison de la boucle par décroissance du variant selon une relation bien fondée ($\prec$). Le second est de garantir le maintien de l'invariant tout au long de l'exécution de la boucle, c'est-à-dire tant que $(C\ e)$ est vrai. Pour remplir ces deux objectifs, il suffit d'imposer la précondition $(\lambda\ e \Rightarrow C\ e\ \wedge\ I\ e\ \wedge\ v = (V\ e))$ et la postcondition $(\lambda\ e' \Rightarrow I\ e'\ \wedge\ (V\ e') \prec v)$ au corps de la boucle S.

Lorsque les trois prémisses précédentes sont satisfaites, cela mène à deux conséquences. Premièrement, la boucle s'arrête si l'état initial e est tel que $(I\ e)$ est vrai. Et, deuxièmement, lorsque la boucle s'arrête sur un état e', on peut compter sur le fait que $\neg(C\ e')\wedge(I\ e')$ est vrai. Donc, par affaiblissement, toute précondition P satisfaisant la quatrième prémisse est une précondition valide de la boucle. De plus, par renforcement, toute postcondition Q satisfaisant la dernière prémisse est une postcondition valide de la boucle.

Sous-programmes. Conformément à l'interprétation relationnelle, le programme $\langle R\rangle$ termine sur l'état initial e si et seulement si $(\exists\ e' \cdot R\ e\ e')$ est vrai, et de plus, pour tout état final e', $(R\ e\ e')$ est vrai. Par conséquent, la prémisse de la règle (sub) signifie littéralement ceci : pour tout état initial e, si $(P\ e)$ est vrai, alors $\langle R\rangle$ termine et tout état final e' est tel que $(Q\ e')$ est vrai. Ce qui correspond bien à la définition d'un triplet de Hoare valide.

4.6.1.2 Triplets de Hoare et sémantique prédicative

Le cadre des triplets de Hoare permet d'exprimer une correspondance entre le programme S et le sous-programme spécifié par l'interprétation prédicative de S, soit $\langle[\![S]\!]\rangle$. Ceci n'est pas surprenant, puisque après tout, exécuter S devrait revenir au même qu'exécuter un sous-programme dont on exige qu'il se comporte conformément à la sémantique de S. Le lemme 10 ci-dessous formalise le fait que S et $\langle[\![S]\!]\rangle$ sont équivalents au sens où ils satisfont les mêmes spécifications.

Lemme 10. *S et $\langle[\![S]\!]\rangle$ satisfont les mêmes spécifications :*

$$\forall\ S\ P\ Q \cdot [P]\ S\ [Q]\ \leftrightarrow\ [P]\ \langle[\![S]\!]\rangle\ [Q]$$

Schéma de preuve. Dans la direction ($\rightarrow$) on procède par induction sur la structure de $\{P\}\ S\ \{Q\}$ donnée dans la figure 4.3. Nous considérons le cas $S =$ **while** C **do** P **end**, qui est le seul cas non trivial. Pour ce cas, l'hypothèse $[P]\ S\ [Q]$ implique l'existence d'une relation bien fondée que nous noterons ($\prec$). La preuve consiste à utiliser les deux principes d'induction associés à ($\prec$), et à la sémantique prédicative de la boucle (propriété (2) du lemme 6 en page 37).

Dans la direction ($\leftarrow$) on procède par induction sur S. Le cas non trivial est celui de l'instruction **while**. Donc considérons le cas $S =$ **while** C **do** P **end**. Nous devons fournir des témoins pour le variant V, la relation bien fondée ($\prec$), et l'invariant I. Comme témoin de V on prendra la fonction identité. Comme témoin de ($\prec$) on prendra ($\prec_P^C$). Enfin, comme témoin de I on prendra l'ensemble des états initiaux tels que la boucle termine sur un état final qui satisfait la postcondition Q, soit :

$$\lambda\ e \Rightarrow (\exists\ e' \cdot [\![\textbf{while}\ C\ \textbf{do}\ P\ \textbf{end}]\!]\ e\ e')\ \wedge\ (\forall\ e' \cdot [\![\textbf{while}\ C\ \textbf{do}\ P\ \textbf{end}]\!]\ e\ e' \rightarrow Q\ e')$$

La preuve consiste ensuite à utiliser les deux principes d'induction associés à ($\prec_P^C$), et à la sémantique prédicative de la boucle (propriété (2) du lemme 6). □

4.6.1.3 Règle dérivées

Certaines règles qu'on trouve habituellement dans les présentations de la logique de Hoare sont absentes de notre présentation en figure 4.3. Nous parlons des règles de la figure 4.4 ci-dessous. Soit celles qui correspondent aux instructions **skip** et **abort**, ainsi que la règle dite de *conséquence*.

En l'absence de la règle (conseq) en particulier, il y a exactement une règle par type d'instruction, ce qui permet de simplifier les raisonnements dirigés par la syntaxe. Toute-

$$\frac{}{[P]\ \textbf{skip}\ [P]}\ (\text{skip}) \qquad \frac{\forall\ e \cdot \neg(P\ e)}{[P]\ \textbf{abort}\ [Q]}\ (\text{abort}) \qquad \frac{\begin{array}{l}\forall\ e \cdot (P\ e) \to (P'\ e) \\ [P']\ S\ [Q'] \\ \forall\ e' \cdot (Q'\ e') \to (Q\ e')\end{array}}{[P]\ S\ [Q]}\ (\text{conseq})$$

Figure 4.4: Règles dérivées

fois, l'absence des règles de la figure 4.4 n'affaiblit pas la logique puisque ces dernières sont dérivables de celle de la figure 4.3 comme nous allons maintenant le montrer.

La règle (**skip**) est conséquente à la règle (effet). En effet, dans le cas où f est l'identité et $Q = P$, la prémisse de (effet) est une tautologie et la conclusion est identique à celle de la règle (skip). Plus formellement, on a :

$$\begin{array}{ll} & [P]\ \textbf{skip}\ [P] \\ \leftarrow & [P]\ (\textbf{effect}\ (\lambda\ e \Rightarrow e))\ [P] \\ \leftarrow & (\forall\ e \cdot P\ e \to P\ e) \\ \leftarrow & \mathsf{True} \end{array}$$

La règle (**abort**) se déduit de la règle (sub) puisque lorsque $R = (\lambda\ e\ e' \Rightarrow \mathsf{False})$, la prémisse de cette règle se réduit à celle de (abort). C'est-à-dire :

$$\begin{array}{ll} & [P]\ \textbf{abort}\ [Q] \\ \leftarrow & [P]\ \langle\lambda\ e\ e' \Rightarrow \mathsf{False}\rangle\ [Q] \\ \leftarrow & (\forall\ e \cdot P\ e \to \mathsf{False}) \\ \leftarrow & (\forall\ e \cdot \neg(P\ e)) \end{array}$$

La règle de (**conseq**) découle des règles (skip) et (séquence). En effet, supposons qu'on a $[P']\ S\ [Q']$. Alors, par (séquence) on a aussitôt $[P]$ **skip;**$S\ [Q']$ si on suppose $[P]$ **skip** $[P']$. Si de plus on suppose $[Q']$ **skip** $[Q]$, on peut à nouveau appliquer (séquence) à $[P]$ **skip;**$S\ [Q']$ pour obtenir $[P]$ **skip;**S**;skip** $[Q]$, qui est équivalent à $[P]\ S\ [Q]$ puisque **skip** est neutre pour (**;**). En termes plus formels, la dérivation est la suivante :

$$\begin{array}{ll} & [P]\ S\ [Q] \\ \leftarrow & [P]\ (\textbf{skip;}S)\textbf{;skip}\ [Q] \\ \leftarrow & [P]\ \textbf{skip;}S\ [Q'] \quad \wedge \quad [Q']\ \textbf{skip}\ [Q] \\ \leftarrow & [P]\ \textbf{skip}\ [P'] \quad \wedge \quad [P']\ S\ [Q'] \quad \wedge \quad [Q']\ \textbf{skip}\ [Q] \\ \leftarrow & (\forall\ e \cdot P\ e \to P'\ e) \quad \wedge \quad [P']\ S\ [Q'] \quad \wedge \quad (\forall\ e' \cdot Q'\ e' \to Q\ e') \end{array}$$

4.6.2 Raffinement en logique de Hoare

Précédemment nous avons mentionné que l'assertion $S_2 \sqsubseteq S_1$ faisait intuitivement référence à la situation dans laquelle toute spécification satisfaite par S_1 l'est aussi par S_2. Si on considère que les spécifications sont des paires (précondition,postcondition), cette définition de la relation de raffinement s'exprime tout naturellement dans le cadre des triplets de Hoare comme indiqué ci-après dans la définition 16. Pour différencier la définition qui suit de la précédente définition de la relation de raffinement, nous utiliserons la notation ($\sqsubseteq_h$).

Définition 16 (Raffinement en logique de Hoare [7])**.**

$$S_2 \sqsubseteq_h S_1 \quad \stackrel{\text{def}}{=} \quad \forall\ P\ Q\ \cdot\ [P]\ S_1\ [Q]\ \rightarrow\ [P]\ S_2\ [Q]$$

Cette définition peut néanmoins interroger dans la mesure où on sait que les triplets de Hoare ne sont pas toujours suffisamment expressifs lorsque la postcondition Q est un prédicat sur l'état final uniquement. Si par exemple on veut exprimer dans la postcondition le fait qu'une variable x voit sa valeur augmenter, il faut avoir accès à la valeur initiale. Pour ce faire, une solution consiste à utiliser la technique des *constantes logiques* [59, p.53]. Pour illustration, considérons l'assertion suivante :

$$\forall\ X \cdot [\ \lambda\ x \Rightarrow X = x\ ]\ x := x + 1\ [\ \lambda\ x' \Rightarrow x' > X\ ]$$

En exigeant que la constante X soit égale à la valeur de x au niveau de la précondition, on a sauvegardé (logiquement parlant) la valeur initiale de x dans X. Ensuite, La quantification universelle sur X fait qu'on peut en faire usage dans la postcondition pour pouvoir exprimer l'augmentation de la valeur de x. Par mesure de simplification, on définit ci-dessous le triplet de Hoare *étendu* qui permet d'éviter la déclaration manuelle des constantes logiques. Dans le triplet de Hoare *étendu*, la postcondition Q est du type $\mathsf{Spec}_{T,T'}$ des relations entre état initial et état final.

Définition 17 (Triplet de Hoare *étendu*)**.** *Le triplet étendu* $[P]\ S\ [[Q]]$ *est un prédicat valide si et seulement si pour tout état initial* e *tel que* $(P\ e)$ *est vrai,* S *termine et l'état final* e' *est tel que* $(Q\ e\ e')$ *est vrai. Soit :*

$$[P]\ S\ [[Q]] \quad \stackrel{\text{def}}{=} \quad \forall\ e_i \cdot [\ \lambda\ e \Rightarrow e_i = e \wedge P\ e\ ]\ S\ [\ \lambda\ e' \Rightarrow Q\ e_i\ e'\ ]$$

Notons au passage que si la postcondition Q ne dépend que de l'état final, la définition ci-dessus se réduit à la définition 15 des triplets de Hoare. On a donc le lemme suivant.

Lemme 11. $\forall\ S\ P\ Q \cdot [P]\ S\ [Q] \quad \leftrightarrow \quad [P]\ S\ [[\ \lambda\ \_\ e' \Rightarrow Q\ e'\ ]]$

Preuve.

$$
\begin{array}{ll}
 & [P]\ S\ [Q] \\
\leftrightarrow & [P]\ \langle [\![S]\!] \rangle\ [Q] \qquad \text{(lemme 10)} \\
\leftrightarrow & \forall\ e \cdot P\ e \rightarrow (\exists\ e' \cdot [\![S]\!]\ e\ e') \wedge \forall\ e' \cdot [\![S]\!]\ e\ e' \rightarrow Q\ e' \qquad \text{(élim. de (sub))} \\
\leftrightarrow & \forall\ e_i\ e \cdot e_i = e \wedge P\ e \rightarrow (\exists\ e' \cdot [\![S]\!]\ e\ e') \wedge \forall\ e' \cdot [\![S]\!]\ e\ e' \rightarrow Q\ e' \\
\leftrightarrow & \forall\ e_i \cdot [\ \lambda\ e \Rightarrow e_i = e \wedge P\ e\ ]\ \langle [\![S]\!] \rangle\ [\ \lambda\ e' \Rightarrow (\lambda\ \_\ e' \rightarrow Q\ e')\ e_i\ e'\ ] \qquad \text{(intro. de (sub))} \\
\leftrightarrow & \forall\ e_i \cdot [\ \lambda\ e \Rightarrow e_i = e \wedge P\ e\ ]\ S\ [\ \lambda\ e' \Rightarrow (\lambda\ \_\ e' \Rightarrow Q\ e')\ e_i\ e'\ ] \qquad \text{(lemme 10)} \\
\leftrightarrow & [P]\ S\ [[\ \lambda\ \_\ e' \Rightarrow Q\ e'\ ]] \qquad \text{(définition 17)}
\end{array}
$$

□

Nous pouvons maintenant définir une relation de raffinement basée sur une notion plus expressive de spécification dans laquelle, la postcondition est de type $\mathsf{Spec}_{T,T'}$. Pour la distinguer des précédentes, nous noterons cette relation par $(\sqsubseteq_H)$.

Définition 18 (Raffinement avec triplets étendus)**.**

$$S_2 \sqsubseteq_H S_1 \quad \stackrel{\text{def}}{=} \quad \forall\ P\ Q\ \cdot\ [P]\ S_1\ [[Q]]\ \rightarrow\ [P]\ S_2\ [[Q]]$$

Bien que les interrogations sur la définition 16 qui ont mené à la définition 18 semblent légitimes, il se fait que les deux définitions sont équivalentes comme le montre le théorème 2 ci-dessous. Par conséquent, pour conduire nos raisonnements méta-théoriques, nous pouvons nous contenter de la première définition qui est la plus simple des deux.

Théorème 2 ($\sqsubseteq_h$ et $\sqsubseteq_H$ sont équivalentes)**.** $\forall\ S_2\ S_1 \cdot S_2 \sqsubseteq_h S_1 \quad \leftrightarrow \quad S_2 \sqsubseteq_H S_1$

Preuve.

Cas $(\rightarrow)$:

$$
\begin{array}{ll}
 & S_2 \sqsubseteq_H S_1 \\
\leftarrow & \forall\ P\ Q\ \cdot\ [P]\ S_1\ [[Q]]\ \rightarrow\ [P]\ S_2\ [[Q]] \qquad \text{(définition 18)} \\
\leftarrow & [P]\ S_1\ [[Q]]\ \rightarrow\ (\forall\ e_i \cdot [\ \lambda\ e \Rightarrow e_i = e \wedge P\ e\ ]\ S_2\ [\ \lambda\ e' \Rightarrow Q\ e_i\ e'\ ]) \qquad \text{(définition 17)}
\end{array}
$$

$$
\begin{array}{ll}
\leftarrow & [P]\ S_1\ [[Q]]\ \rightarrow\ (\forall\ e_i \cdot [\ \lambda\ e \Rightarrow e_i = e \wedge P\ e\ ]\ S_1\ [\ \lambda\ e' \Rightarrow Q\ e_i\ e'\ ]) \\
 & \wedge\ \ S_2 \sqsubseteq_h S_1
\end{array}
$$

$$
\begin{array}{ll}
\leftarrow & [P]\ S_1\ [[Q]]\ \rightarrow\ [P]\ S_1\ [[Q]] \quad \wedge \quad S_2 \sqsubseteq_h S_1 \qquad \text{(définition 17)} \\
\leftarrow & S_2 \sqsubseteq_h S_1
\end{array}
$$

Cas ($\leftarrow$) :

$$
\begin{array}{lll}
& S_2 \sqsubseteq_h S_1 & \\
\leftarrow & \forall\ P\ Q\ \cdot\ [P]\ S_1\ [Q]\ \rightarrow\ [P]\ S_2\ [Q] & \text{(définition 16)} \\
\leftarrow & [P]\ S_1\ [Q]\ \rightarrow\ [P]\ S_2\ [[\ \lambda\ \_\ e' \Rightarrow Q\ e'\ ]] & \text{(lemme 11)} \\
\leftarrow & ([P]\ S_1\ [Q]\ \rightarrow\ [P]\ S_1\ [[\ \lambda\ \_\ e' \Rightarrow Q\ e'\ ]])\ \wedge\ S_2 \sqsubseteq_H S_1 & \\
\leftarrow & ([P]\ S_1\ [Q]\ \rightarrow\ \forall\ e_i \cdot [\ \lambda\ e \Rightarrow e_i = e \wedge P\ e\ ]\ S_1\ [Q])\ \wedge\ S_2 \sqsubseteq_H S_1 & \text{(définition 17)} \\
\leftarrow & ([P]\ S_1\ [Q]\ \rightarrow\ [P]\ S_1\ [Q])\ \wedge\ S_2 \sqsubseteq_H S_1 & \text{(règle (conseq))} \\
\leftarrow & S_2 \sqsubseteq_H S_1 &
\end{array}
$$

□

4.6.3 Équivalence avec le point de vue relationnel

Si nous comparons les deux points de vue, il est clair que le point de vue à la Hoare (définition 16) nous offre une définition du raffinement plus intuitive que la définition 14 utilisée dans le point de vue relationnel. Étant donné que dans un cas les programmes sont interprétés comme des relations entre états de programme, et que dans l'autre cas ils sont interprétés comme des relations entre ensembles d'états, il n'est pas possible de formuler un critère d'équivalence direct entre les deux interprétations. Néanmoins, nous pouvons formuler le critère d'équivalence indirect qui consiste à considérer que deux interprétations des programmes sont équivalentes si tout raffinement valide dans une interprétation est aussi valide dans l'autre interprétation. Avec le théorème 3 ci-dessous, nous montrons que, vis-à-vis de ce critère, les deux points de vue sont logiquement équivalents.

Théorème 3 ($\sqsubseteq$ et $\sqsubseteq_h$ sont équivalentes)**.** $\forall\ S_1\ S_2\ \cdot\ S_2 \sqsubseteq S_1 \quad \leftrightarrow \quad S_2 \sqsubseteq_h S_1$

Preuve.

Cas ($\rightarrow$) :

$$
\begin{array}{lll}
& S_2 \sqsubseteq_h S_1 & \\
\leftarrow & \forall\ P\ Q\ \cdot\ [P]\ S_1\ [Q]\ \rightarrow\ [P]\ S_2\ [Q] & \text{(définition 16)} \\
\leftarrow & [P]\ \langle [\![S_1]\!] \rangle\ [Q]\ \rightarrow\ [P]\ \langle [\![S_2]\!] \rangle\ [Q] & \text{(lemme 10)} \\
\leftarrow & \begin{array}{l} (\forall\ e \cdot P\ e\ \rightarrow\ (\exists\ e' \cdot [\![S_1]\!]\ e\ e') \wedge \forall\ e' \cdot [\![S_1]\!]\ e\ e' \rightarrow Q\ e') \\ \rightarrow (\forall\ e \cdot P\ e\ \rightarrow\ (\exists\ e' \cdot [\![S_2]\!]\ e\ e') \wedge \forall\ e' \cdot [\![S_2]\!]\ e\ e' \rightarrow Q\ e') \end{array} & \text{(élim. de (sub))} \\
\leftarrow & \begin{array}{l} \forall\ e \cdot (\exists\ e' \cdot [\![S_1]\!]\ e\ e') \wedge \forall\ e' \cdot [\![S_1]\!]\ e\ e' \rightarrow Q\ e' \\ \quad \rightarrow (\exists\ e' \cdot [\![S_2]\!]\ e\ e') \wedge \forall\ e' \cdot [\![S_2]\!]\ e\ e' \rightarrow Q\ e' \end{array} &
\end{array}
$$

$$\leftarrow \quad \forall\ e \cdot (\exists\ e' \cdot [\![S_1]\!]\ e\ e') \\ \quad \rightarrow\ (\exists\ e' \cdot [\![S_2]\!]\ e\ e') \wedge \forall\ e' \cdot [\![S_2]\!]\ e\ e' \rightarrow [\![S_1]\!]\ e\ e'$$

$\leftarrow \quad S_2 \sqsubseteq S_1$ (définition 14)

Cas ($\leftarrow$) :

$S_2 \sqsubseteq S_1$

$$\leftarrow \quad \forall\ e \cdot (\exists\ e' \cdot [\![S_1]\!]\ e\ e') \\ \quad \rightarrow\ (\exists\ e' \cdot [\![S_2]\!]\ e\ e') \wedge \forall\ e' \cdot [\![S_2]\!]\ e\ e' \rightarrow [\![S_1]\!]\ e\ e'$$ (définition 14)

$$\leftarrow \quad (\forall\ e_i \cdot e_i = e \rightarrow (\forall\ e' \cdot [\![S_1]\!]\ e_i\ e' \rightarrow [\![S_1]\!]\ e\ e') \wedge (\exists\ e' \cdot [\![S_1]\!]\ e_i\ e')) \\ \rightarrow \forall\ e_i \cdot e_i = e \rightarrow (\exists\ e' \cdot [\![S_2]\!]\ e_i\ e') \wedge \forall\ e' \cdot [\![S_2]\!]\ e_i\ e' \rightarrow [\![S_1]\!]\ e\ e')$$

$$\leftarrow \quad [\ \lambda\ e_i \Rightarrow e_i = e\ ]\ \langle [\![S_1]\!] \rangle\ [\ \lambda\ e' \Rightarrow [\![S_1]\!]\ e\ e'\ ] \\ \rightarrow [\ \lambda\ e_i \Rightarrow e_i = e\ ]\ \langle [\![S_2]\!] \rangle\ [\ \lambda\ e' \Rightarrow [\![S_1]\!]\ e\ e'\ ]$$ (intro. de (sub))

$$\leftarrow \quad [\ \lambda\ e_i \Rightarrow e_i = e\ ]\ S_1\ [\ \lambda\ e' \Rightarrow [\![S_1]\!]\ e\ e'\ ] \\ \rightarrow [\ \lambda\ e_i \Rightarrow e_i = e\ ]\ S_2\ [\ \lambda\ e' \Rightarrow [\![S_1]\!]\ e\ e'\ ]$$ (lemme 10)

$$\leftarrow \quad \forall\ P\ Q\ \cdot\ [P]\ S_1\ [Q]\ \rightarrow\ [P]\ S_2\ [Q]$$

$\leftarrow \quad S_2 \sqsubseteq_h S_1$ (définition 16)

□

4.7 Conclusion

Dans ce chapitre, nous avons présenté une sémantique des programmes impératifs qui permet d'interpréter ces derniers comme des relations binaires entres états de programme. Dans le même esprit que Sekerinski [70], et à la différence des travaux similaires tels que par exemple ceux dus à Frappier [30] et Tchier [75], ces relations sont ici représentées par des prédicats dans le cadre de la théorie des types plutôt que dans le calcul des relations. Aussi, Sekerinski, Frappier et Tchier utilisent la relation de raffinement pour définir la sémantique des boucles, sachant que cette même sémantique peut servir à prouver des raffinements. Notre approche se différencie par le fait que nous nous contentons de la relation basique d'inclusion des spécifications pour définir la sémantique des boucles. Par conséquent, la sémantique qui nous permet de définir la notion de raffinement est indépendante de cette dernière. À juste titre, ceci place explicitement la notion de raffinement à un niveau d'abstraction plus élevé.

Nous avons aussi présenté, dans le cadre de la théorie des types, une sémantique axiomatique sous la forme d'une logique de Hoare, et établi que les deux approches sont équivalentes dans le sens où tout raffinement valide dans une approche l'est aussi dans l'autre. Ceci montre qu'en matière de formalisation de sémantiques en théorie des types, l'approche relationnelle est une alternative crédible aux approches basées sur la logique de Hoare. De plus, la sémantique prédicative que nous avons formalisé se limite à donner un sens mathématique aux programmes permettant ainsi d'envisager séparément le problème de la preuve de programmes. En comparaison, les sémantiques basées sur la logique de Hoare vont tout de suite plus loin, puisque de fait, ce sont aussi, et peut être avant tout, des logiques de preuves de programmes.

Chapitre 5

Méthodes de preuve de raffinements

Dans ce chapitre, nous nous intéressons aux techniques de preuve permettant d'établir en général la validité d'un raffinement entre deux programmes. En particulier, lorsque qu'une spécification S est donnée sous la forme d'un prédicat *avant-après*, prouver que S est satisfaite par un programme P consiste à établir que l'assertion $(P \sqsubseteq \langle S \rangle)$ est valide. L'approche relationnelle du raffinement de programmes se veut plus uniforme que les autres approches et traite les problèmes de raffinements de programmes en les ramenant systématiquement, au moins au niveau théorique, à des problèmes de raffinements de spécifications. En pratique, il s'ensuit une première difficulté puisque la traduction des programmes en spécifications peut aboutir à des expressions logiques difficiles à appréhender et à manipuler. Eu égard à notre choix de sémantique, l'opérateur de point fixe lfp apparaît dans les expressions logiques en question dès que les programmes impliqués contiennent des boucles. Il y a là une deuxième difficulté puisque l'opérateur lfp est un opérateur du second ordre qui quantifie universellement sur des spécifications. Nous illustrerons plus en détails ces deux difficultés dans la section 5.1. Ensuite nous nous intéresserons à la simplification des formules qui résultent de la traduction des problèmes de raffinement de programmes en problèmes de raffinement de spécifications. D'abord, dans la section 5.2 nous présentons une interprétation prédicative plus évoluée qui produit des formules plus simples que notre interprétation précédente. Ensuite, dans la section 5.3 nous présentons un calcul qui est analogue au calcul de la plus faible précondition, et qui, comme ce dernier, permet d'automatiser certains raisonnements logiques de routines qui peuvent s'avérer assez fastidieux. Enfin, dans la section 5.4 nous examinons plus spécifiquement le cas des boucles, en particulier nous formulons des conditions nécessaires et suffisantes permettant de valider un raffinement de boucle en logique du premier ordre.

5.1 Motivation

Dans cette section, nous illustrons les difficultés liées aux preuves déductives de raffinements. Nous examinons d'abord le cas des programmes sans boucles, et nous nous penchons ensuite sur le cas des boucles.

5.1.1 Raffinements des programmes sans boucles

Considérons l'exemple suivant dans lequel nous montrons, à partir des définitions précédentes, que deux programmes très simples sont équivalents.

Exemple 7 (Équivalence de programmes)**.** *Soient P et Q les deux programmes suivants :*

$$P \stackrel{\text{def}}{=} x := 1 \,;\, y := x + 2 \qquad Q \stackrel{\text{def}}{=} x := 1 \,;\, y := 3$$

Pour preuve que P et Q sont équivalents, on a la dérivation suivante :

$$\begin{aligned}
& [\![P]\!] \\
\leftrightarrow\ & (\exists\ x_a\ y_a \cdot x_a = 1\ \wedge\ y_a = y\ \wedge\ x' = x_a\ \wedge\ y' = x_a + 2) \\
& \quad \wedge\ \forall\ x_a\ y_a \cdot x_a = 1\ \wedge\ y_a = y\ \rightarrow\ \exists\ x'\ y' \cdot x' = x_a\ \wedge\ y' = x_a + 2 \\
\leftrightarrow\ & (\exists\ x_a\ y_a \cdot x_a = 1\ \wedge\ y_a = y\ \wedge\ x' = 1\ \wedge\ y' = 1 + 2) \\
& \quad \wedge\ \forall\ x_a\ y_a \cdot x_a = 1\ \wedge\ y_a = y\ \rightarrow\ \exists\ x'\ y' \cdot x' = 1\ \wedge\ y' = 1 + 2 \\
\leftrightarrow\ & (\exists\ x_a\ y_a \cdot x_a = 1\ \wedge\ y_a = y\ \wedge\ x' = 1\ \wedge\ y' = 3) \\
& \quad \wedge\ \forall\ x_a\ y_a \cdot x_a = 1\ \wedge\ y_a = y\ \rightarrow\ \exists\ x'\ y' \cdot x' = 1\ \wedge\ y' = 3 \\
\leftrightarrow\ & [\![Q]\!]
\end{aligned}$$

Il était clair au départ que P et Q étaient équivalents puisqu'ils sont tous les deux équivalents à $\langle x' = 1 \wedge y' = 3 \rangle$. Pourtant, comme le montre l'exemple 7 ci-dessus, lorsque nous cherchons à le prouver en appliquant la fonction $[\![\cdot]\!]$, nous devons passer par un certain nombre de raisonnements certes évidents mais tout de même assez mécaniques. En effet, l'expression $[\![P]\!]$ est difficile à appréhender en raison de sa taille qui est beaucoup plus importante que celle de P. En fait, si nous examinons la situation plus généralement, on verra que l'expression $[\![\langle P_1 \rangle ; \langle P_2 \rangle ; \ldots \langle P_n \rangle]\!]$ est de taille exponentielle en n après expansion des définitions, ce qui est évidemment peu attractif. De plus, la présence dans cette expression d'un grand nombre de quantificateurs dûs à l'opérateur $\Box$ rendent sa lecture

d'autant plus complexe. Néanmoins, dans le cas de l'exemple 7, des raisonnements simples et mécaniques permettent de montrer que P et Q sont équivalents à l'expression $\langle x' = 1 \ \wedge \ y' = 3 \rangle$. Il serait donc utile de pouvoir, dans la mesure du possible, automatiser de tels raisonnements car cela permet de réduire l'effort de preuve. Un moyen d'automatiser certains raisonnements consiste à faire appel au wp-calcul comme nous l'illustrons dans l'exemple ci-dessous.

Exemple 8 (Preuve d'équivalence avec le wp-calcul)**.** *Considérons les programmes suivants :*

$$S \stackrel{\text{def}}{=} x := x - y; y := x + y; x := y - x \qquad T \stackrel{\text{def}}{=} x := x + y; y := x - y; x := x - y$$

Par la dérivation suivante, on montre que S et T sont équivalents.

$$\begin{aligned}
& S \equiv T \\
\leftrightarrow \ & \forall \ P \ Q \cdot \{P\} \ S \ \{Q\} \leftrightarrow \{P\} \ T \ \{Q\} \\
\leftrightarrow \ & \forall \ P \ Q \cdot (\forall \ x \ y \cdot (P \ x \ y) \rightarrow (\mathit{wp} \ S \ Q) \ x \ y) \leftrightarrow (\forall \ x \ y \cdot (P \ x \ y) \rightarrow (\mathit{wp} \ T \ Q) \ x \ y) \\
\leftrightarrow \ & \forall \ Q \cdot \forall \ x \ y \cdot (\mathit{wp} \ S \ Q) \ x \ y \leftrightarrow (\mathit{wp} \ T \ Q) \ x \ y \\
\leftrightarrow \ & Q \ ((x - y + y) - (x - y)) \ (x - y + y) \ \leftrightarrow \ Q \ ((x + y) - (x + y - y)) \ (x + y - y) \\
\leftrightarrow \ & Q \ y \ x \ \leftrightarrow \ Q \ y \ x \\
\leftrightarrow \ & \mathit{True}
\end{aligned}$$

Grâce à la nature calculatoire du transformateur de prédicats wp, on obtient des formules logiques plus simples. Néanmoins, nous devons travailler avec des formules du second ordre bien que l'équivalence à prouver concerne des programmes du premier ordre. Pour ces derniers, il serait utile de pouvoir travailler avec des formules simples mais du premier ordre. En effet, on pourrait alors invoquer des procédures de décision spécialisées qui permettent par exemple de traiter des problèmes du premier ordre en arithmétique linéaire, limitant ainsi les interventions manuelles.

5.1.2 Raffinements de boucles

Pour prouver qu'une boucle raffine une spécification, il est en pratique plus simple de travailler avec une formule logique du premier ordre qui caractérise la boucle. En général les notions d'invariant et de variant sont utilisées pour caractériser les boucles en logique du premier ordre, et la plupart des formalisations des théories de raffinement s'appuient sur ces notions. En effet, considérons la boucle (**while** C **do** P **end**). Supposons que nous disposions d'un invariant du premier ordre I, d'un variant V et d'une relation bien fondée ($\prec$). Enfin, soit l'assertion suivante :

$$A_w \stackrel{\text{def}}{=} \forall\, e_i \cdot [\, \lambda\, e \Rightarrow C\ e \ \wedge\ I\ e_i\ e\,]\ P\ [[\, \lambda\, e\ e' \Rightarrow I\ e_i\ e' \ \wedge\ (V\ e') \prec (V\ e)\,]]$$

Cette formule affirme que le corps de la boucle maintient l'invariant I et fait décroître le variant V, sa validité implique que la boucle (**while** C **do** P **end**) raffine la spécification suivante :

$$I_w \stackrel{\text{def}}{=} \langle \lambda\ e\ e' \Rightarrow I\ e\ e' \ \wedge\ \neg(C\ e') \rangle$$

Dès lors, par transitivité, toute spécification raffinée par I_w, l'est aussi par la boucle. Nous pouvons donc utiliser cette spécification du premier ordre pour raisonner à propos de la boucle. La condition A_w revient en fait au raffinement suivant :

$$\forall\, e_i \cdot P \ \sqsubseteq\ \langle \lambda\ e\ e' \Rightarrow I\ e_i\ e \ \rightarrow\ I\ e_i\ e' \wedge (V\ e') \prec (Ve) \rangle$$

Dans cette formule, on voit que la spécification a une forme particulière. Par exemple, le membre de gauche ($I\ e_i\ e$) de l'implication et la première partie du membre de droite ($I\ e_i\ e'$) ont la même forme. Ainsi, l'utilisation des notions d'invariant et de variant revient à contraindre la forme des spécifications associées aux corps des boucles, alors que évidemment toute autre spécification logiquement équivalente ferait l'affaire, quelle que soit sa forme. Dans leur quête d'uniformité, les interprétations relationnelles des programmes proposent une caractérisation alternative des boucles, toujours en logique du premier ordre, mais s'appuyant sur la seule notion générale de spécification pour abstraire les corps des boucles. Nous nous proposons d'étudier et de formaliser cette caractérisation alternative dans le cadre de la théorie des types.

5.2 Une interprétation prédicative alternative

Dans cette section, nous étudions l'interprétation des programmes de la forme $S;\langle T \rangle$ pour chaque type d'instruction S à la recherche d'opportunités de simplification des expressions

$[\![S;\langle T\rangle]\!]$. Dans un premier temps, ceci nous amène à définir une nouvelle interprétation de $S;\langle T\rangle$. Nous montrons ensuite que cette nouvelle interprétation est équivalente à l'interprétation précédente $[\![\cdot]\!]$, nous permettant ainsi de préserver les résultats qui ont précédemment été formulés dans celle-ci.

5.2.1 Interprétation spécifique de la composition séquentielle

Afin de simplifier les formules logiques apparaissant durant les preuves de raffinements, nous éliminons l'opérateur $\Box$ de ces formules à chaque fois que cela est possible sur une base syntaxique. Par exemple, nous savons que l'équivalence ci-dessous est valide :

$$i := i + 1;\langle \lambda\ i\ i' \Rightarrow R\ i\ i'\rangle \ \equiv\ \langle \lambda\ i\ i' \Rightarrow R\ (i+1)\ i'\rangle$$

Dans les cas où cela est possible, nous aimerions calculer la seconde expression automatiquement en fonction des deux arguments de la composition séquentielle. Pour ce faire, nous examinons et simplifions l'expression $[\![S;\langle R\rangle]\!]$ pour chaque type d'instruction S. Ce processus aboutit à la fonction sémantique $[\![\cdot,\cdot]\!]$ présentée dans la figure 5.1 ci-dessous.

$$
\begin{array}{ll}
[\![\textbf{effect}\ f,\ T]\!] & \stackrel{\text{def}}{=} \lambda\ e\ e' \Rightarrow T\ (f\ e)\ e' \\
[\![\langle T_1\rangle,\ T]\!] & \stackrel{\text{def}}{=} T_1 \sqsupset T \\
[\![\textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end},\ T]\!] & \stackrel{\text{def}}{=} [\![S_1,\ T]\!] \lhd C \rhd [\![S_2,\ T]\!] \\
[\![S_1;S_2,\ T]\!] & \stackrel{\text{def}}{=} [\![S_1,\ [\![S_2,\ T]\!]]\!] \\
[\![\textbf{while}\ C\ \textbf{do}\ S\ \textbf{end},\ T]\!] & \stackrel{\text{def}}{=} (\mathsf{lfp}\ (\lambda\ \mathcal{X} \Rightarrow [\![S,\ \mathcal{X}]\!] \lhd C \rhd [\![\textbf{skip}]\!])) \sqsupset T
\end{array}
$$

Figure 5.1: Une interprétation prédicative de $S;\langle T\rangle$

À partir d'un programme S et d'une spécification T, $[\![S,\ T]\!]$ donne une interprétation prédicative aux programmes de la forme $S;\langle T\rangle$ qui est équivalente à la précédente interprétation $[\![S;\langle T\rangle]\!]$. Le type de la fonction $[\![\cdot,\cdot]\!]$ est donc :

$$\mathsf{Statement}^{T,U} \to \mathsf{Spec}_{U,V} \to \mathsf{Spec}_{T,V}$$

À présent, détaillons la définition de $[\![\cdot,\cdot]\!]$ pour chaque type de programme S.

Affectation. Lorsque S est de la forme (**effect** f), nous pouvons tirer parti du fait que f est une fonction totale. Nous sommes donc dans le cas particulier où $[\![S;\langle T\rangle]\!]$ se simplifie en $(\lambda\ e\ e' \Rightarrow T\ (f\ e)\ e')$. En effet, exécuter $S;\langle T\rangle$ dans l'état initial e revient à exécuter $\langle T\rangle$ dans un état initial qui correspond à l'image de e par f.

Sous-programmes. Dans le cas où S est un appel à un sous-programme spécifié par T_1, l'opérateur $\Box$ ne peut être éliminé en général car T_1 est un prédicat arbitraire qui n'offre aucune propriété particulière qui pourrait être exploitée. Par conséquent, $[\![S,\ T]\!]$ est identique à $[\![S;\langle T\rangle]\!]$.

Alternative. Pour simplifier $[\![S;\langle T\rangle]\!]$ dans le cas où S est une instruction de type if-then-else on part de l'équivalence suivante :

$$(\textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end});\langle T\rangle \quad \equiv \quad \textbf{if } C \textbf{ then } S_1;\langle T\rangle \textbf{ else } S_2;\langle T\rangle \textbf{ end}$$

Étant donnée cette équivalence, la simplification $[\![S,\ T]\!]$ de $[\![S;\langle T\rangle]\!]$ se calcule récursivement à partir des simplifications de $[\![S_1;\langle T\rangle]\!]$ et $[\![S_2;\langle T\rangle]\!]$.

Séquence. Comme la composition séquentielle de programmes est une opération associative, l'expression $[\![(S_1;S_2);\langle T\rangle]\!]$ s'écrit aussi $[\![S_1;(S_2;\langle T\rangle)]\!]$. La simplification peut donc procéder en deux étapes. D'abord on simplifie $[\![S_2;\langle T\rangle]\!]$ en calculant récursivement $[\![S_2,\ T]\!]$. Ensuite, il s'agit de simplifier $[\![S_1;\langle[\![S_2,\ T]\!]\rangle]\!]$ en calculant $[\![S_1,\ [\![S_2,\ T]\!]]\!]$.

Itération. Le cas des boucles est le second cas où l'opérateur $\Box$ ne peut pas être complètement éliminé en général. En effet, la spécification calculée par lfp peut se comprendre en termes ensemblistes comme une intersection généralisée. Or, l'opérateur $\Box$ n'est pas distributive par rapport à l'intersection. Donc, nous devons le laisser à l'extérieur. L'opérateur $\Box$ a néanmoins été éliminé de l'argument de l'opérateur lfp. Ceci est correct puisque si on suppose que $[\![S,\ \mathcal{X}]\!]$ calcule bien une simplification de l'expression $[\![S;\langle\mathcal{X}\rangle]\!]$ qui est équivalente à cette dernière, alors on peut conclure que l'argument de lfp est équivalent à la fonction $\mathcal{W}_S^C$ de la figure 4.2 page 34.

5.2.2 Équivalence des interprétations prédicatives

L'interprétation $[\![\cdot,\cdot]\!]$ est plus complexe que l'interprétation $[\![\cdot]\!]$. Néanmoins, $[\![\cdot,\cdot]\!]$ est beaucoup moins dépendante de l'opérateur $\Box$. Dans un contexte de preuve, il est donc plus pratique d'utiliser $[\![\cdot,\cdot]\!]$ d'autant plus que les opérateurs de programmation peuvent être utilisés dans l'écriture de spécifications. Bien entendu, nous aurons besoin de transférer les résultats généraux déjà établis dans l'interprétation $[\![\cdot]\!]$. Par conséquent, nous devons formellement établir l'équivalence des deux interprétations.

Fondamentalement, $[\![S,\ T]\!]$ calcule une expression équivalente à $[\![S;\langle T\rangle]\!]$. Ceci est formalisé par le lemme suivant.

Lemme 12. $\forall\ S\ T \cdot [\![S;\langle T\rangle]\!]\ \equiv\ [\![S,\ T]\!]$

Schéma de preuve. La preuve est une simple preuve par induction sur S. □

À partir de ce lemme, on a le théorème suivant qui établit formellement l'équivalence des nos deux interprétations.

Théorème 4 ($[\![\cdot]\!]$ et $[\![\cdot,\cdot]\!]$ sont logiquement équivalents)**.** $\forall\ S \cdot [\![S]\!]\ \equiv\ [\![S,\ [\![\mathbf{skip}]\!]]\!]$

Preuve. Par le lemme 12, on a $[\![S,\ [\![\mathbf{skip}]\!]]\!]\ \equiv\ [\![S;\mathbf{skip}]\!]$. De plus, on a $[\![S;\mathbf{skip}]\!]\ \equiv\ [\![S]\!]$ puisque $S \equiv S;\mathbf{skip}$. Donc on bien $[\![S]\!]\ \equiv\ [\![S, [\![\mathbf{skip}]\!]]\!]$. □

En passant par le théorème 4 nous pouvons désormais construire des preuves formelles de raffinement entre certains programmes en logique du premier ordre et avec des formules logiques plus simples grâce à la fonction $[\![\cdot,\cdot]\!]$. En fait, la fonction $[\![\cdot,\cdot]\!]$ élimine complètement l'opérateur □ pour tous les programmes dans lesquels les boucles ou les appels de sous-programmes n'apparaissent pas à gauche de l'opérateur (;). Il est important de garder ce fait en tête lorsque les opérateurs de programmation sont utilisés dans l'écriture des spécifications. L'exemple ci-dessous illustre l'utilisation de la fonction $[\![\cdot,\cdot]\!]$ pour prouver l'équivalence entre deux programmes. On notera en particulier que les étapes qui suivent l'application du théorème 4 sont aisément automatisables, car elles sont soit mécaniques soit limitées à des raisonnements en arithmétique linéaire.

Exemple 9 (Preuve d'équivalence au premier ordre)**.** *Soient les deux programmes suivants :*

$$S \stackrel{\text{def}}{=} x := x - y; y := x + y; x := y - x \qquad T \stackrel{\text{def}}{=} x := x + y; y := x - y; x := x - y$$

L'équivalence de S et T est établie par la dérivation suivante.

$$\begin{array}{ll}
& S \equiv T \\
\leftrightarrow & [\![S]\!]\ \equiv\ [\![T]\!] \\
\leftrightarrow & [\![S, [\![\mathbf{skip}]\!]]\!]\ \equiv\ [\![T, [\![\mathbf{skip}]\!]]\!] & (\textit{théorème } 4) \\
\leftrightarrow & [\![x := x - y; y := x + y; x := y - x, [\![\mathbf{skip}]\!]]\!]\ \leftrightarrow\ [\![T, [\![\mathbf{skip}]\!]]\!] \\
\leftrightarrow & y' = x - y + y \wedge x' = (x - y + y) - (x - y)\ \leftrightarrow\ [\![T, [\![\mathbf{skip}]\!]]\!] \\
\leftrightarrow & y' = x \wedge x' = y\ \leftrightarrow\ y' = (x + y) - y \wedge x' = (x + y) - (x + y - y) \\
\leftrightarrow & y' = x \wedge x' = y\ \leftrightarrow\ y' = x \wedge x' = y \\
\leftrightarrow & \mathit{True}
\end{array}$$

5.3 Le calcul de la plus faible pré-spécification

Lorsque les programmes sont interprétés comme des transformateurs de prédicats, le calcul de la plus faible précondition [25] (ou wp-calcul) permet de transformer un problème de raffinement de programmes en un problème de validité d'une formule logique. Ce processus de transformation a en particulier l'avantage d'automatiser certains raisonnements simples. En particulier, les raisonnements sur l'opérateur $\square$ sont sources de complexité. Par exemple, considérons l'assertion $(\langle S_1\rangle;\langle S_2\rangle;...;\langle S_n\rangle) \sqsubseteq \langle R\rangle$. Si nous cherchons à prouver cette assertion directement à partir de notre définition du raffinement de programmes (définition 14 en page 40), nous calculerons une formule dont la taille est exponentielle en n, et nécessitant des raisonnements assez long autour de l'opérateur $\square$. Ceci reste vrai même si nous appliquons le théorème 4 pour utiliser l'interprétation alternative $[\![\cdot,\cdot]\!]$ au lieu de l'interprétation $[\![\cdot]\!]$.

Dans le point de vue relationnel, Le concept de plus faible pré-spécification [36, 23, 41] généralise la notion de plus faible précondition. Afin que notre formalisation soit la plus générale possible, nous l'asseyons donc sur la notion de plus faible pré-spécification plutôt que sur celle de plus faible précondition. Les programmes sont alors interprétés, non plus comme des transformateurs de prédicats, mais plutôt comme des transformateurs de spécifications. Dans cette section, nous formalisons d'abord le concept de plus faible pré-spécification ainsi que le calcul associé qui est analogue au wp-calcul. Ensuite, nous formalisons les résultats principaux qui relient ce calcul à la notion de raffinement. Enfin, nous illustrons brièvement l'utilisation du calcul formalisé à des fins de preuve de raffinements.

5.3.1 Notion de plus faible pré-spécification

La plus faible pré-spécification de R_2 vis-à-vis de R_1 est la spécification la plus permissive K telle que l'assertion $\langle K\rangle;\langle R_2\rangle \sqsubseteq \langle R_1\rangle$ est vraie. Ici, on considère qu'une spécification K_1 est plus permissive qu'une spécification K_2 si K_2 raffine K_1. En quelque sorte, si on imagine que $(;)$ correspond à une multiplication, on obtient K en divisant $\langle R_1\rangle$ par $\langle R_2\rangle$ [23]. La plus faible pré-spécification n'existe pas toujours. Par exemple, il n'existe pas de pré-spécification K telle que l'assertion $\langle K\rangle;\langle x' = 0\rangle \sqsubseteq \langle x' = 1\rangle$ est vraie. Si la plus faible pré-spécification existe, sa formulation prédicative en fonction de R_2 et R_1 est donnée par la définition suivante.

Définition 19 (Plus faible pré-spécification [41]). *Soient R_2 et R_1 deux spécifications. La plus faible pré-spécification de R_2 vis-à-vis de R_1 (notée $(\kappa\ R_2\ R_1)$) est définie comme suit :*

$$\kappa\ R_2\ R_1 \quad \stackrel{\text{def}}{=} \quad \lambda\ e\ e' \Rightarrow (\exists\ e'' \cdot R_2\ e'\ e'')\ \wedge\ (\forall\ e'' \cdot R_2\ e'\ e''\ \rightarrow\ R_1\ e\ e'')$$

Intuitivement, cette définition peut se comprendre de la manière suivante. Considérons un état initial e et un état final e' de $\langle \kappa\ R_2\ R_1 \rangle$. On peut alors dire que l'assertion $(\kappa\ R_2\ R_1\ e\ e')$ est vraie. Dans ce cas, par définition, $\langle R_2 \rangle$ termine sur e'. De plus, toujours par définition, tout état final e'' de $\langle R_2 \rangle$ est tel que (e, e'') est un comportement admissible de $\langle R_1 \rangle$, c'est-à-dire $(R_1\ e\ e'')$ est vraie. En d'autres termes, la composition séquentielle de $\langle \kappa\ R_2\ R_1 \rangle$ avec $\langle R_2 \rangle$ engendre nécessairement des comportements (e, e'') qui sont admissibles pour $\langle R_1 \rangle$.

Exemple 10 (Exemple de calcul de pré-spécification). *Considérons un espace des états de programmes composé des trois états a, b et c. Alors, on calcule la plus faible pré-spécification de $\{(b,c)\}$ vis-à-vis de $\{(a,c)\}$ comme suit :*

$$\begin{aligned}
& \kappa\ \{(b,c)\}\ \{(a,c)\} \\
\equiv\ & \kappa\ (\lambda\ e\ e' \Rightarrow e = b \wedge e' = c)\ (\lambda\ e\ e' \Rightarrow e = a \wedge e' = c) \\
\equiv\ & \lambda\ e\ e' \Rightarrow (\exists\ e'' \cdot e' = b \wedge e'' = c)\ \wedge\ (\forall\ e'' \cdot e' = b \wedge e'' = c\ \rightarrow\ e = a \wedge e'' = c) \\
\equiv\ & \lambda\ e\ e' \Rightarrow e' = b\ \wedge\ (e' = b \rightarrow e = a) \\
\equiv\ & \lambda\ e\ e' \Rightarrow e' = b\ \wedge\ e = a \\
\equiv\ & \{(a,b)\}
\end{aligned}$$

Et on a bien $\langle\{(a,b)\}\rangle;\langle\{(b,c)\}\rangle \sqsubseteq \langle\{(a,c)\}\rangle$.

On peut remarquer que l'opérateur κ permet de formuler la notion de raffinement comme le montre le lemme ci-dessous.

Lemme 13 (Raffinement de spécifications avec κ [23]).

$$\forall\ R_2\ R_1 \cdot \langle R_2 \rangle \sqsubseteq \langle R_1 \rangle \quad \leftrightarrow \quad \forall\ e\ \cdot (\exists\ e' \cdot R_1\ e\ e')\ \rightarrow\ (\kappa\ R_2\ R_1\ e\ e)$$

Preuve. Immédiat à partir des définitions. □

Notons que dans ce lemme, $(\kappa\ R_2\ R_1\ e\ e)$ représente l'ensemble des états e tels que $\langle R_2 \rangle$ termine sur l'état initial e, et tous les états finaux e' de $\langle R_2 \rangle$ à partir de e satisfont la postcondition R_1, et sont donc tels que $(R_1\ e\ e')$ est vraie. Par conséquent, on peut considérer que $(\kappa\ R_2\ R_1\ e\ e)$ calcule la plus faible précondition de R_2 vis-à-vis de la postcondition R_1. En d'autres termes, le lemme 13 est une formulation de la notion de

raffinement de programmes selon le patron bien connu ($P \subseteq (\mathsf{wp}\ S\ Q)$) qui exprime le fait qu'un programme S satisfait une spécification (P, Q). En effet, il suffit de prendre $\langle R_2 \rangle$ pour S, le domaine de R_1 pour la précondition P, et R_1 pour la postcondition Q. Ainsi, la notion de plus faible pré-spécification permet de dériver la notion de plus faible précondition, et c'est en cela qu'elle est plus générale que cette dernière.

5.3.2 Transformateur de spécifications

Si nous appliquons κ à $[\![S]\!]$ pour chaque type d'instruction S, et qu'ensuite nous simplifions les expressions résultantes, nous obtenons un transformateur de spécifications défini par induction sur la syntaxe. Ce transformateur de spécifications est analogue au transformateur de prédicats wp, nous le noterons wpr. À partir d'un programme S et d'une spécification R, (wpr S R) calcule une formule logique équivalente à (κ $[\![S]\!]$ R), et plus simple que cette dernière. Le transformateur wpr est défini ci-dessous.

Définition 20. *On définit le transformateur de spécifications* wpr *dont le type est :*

$$\mathit{Statement}^{U,V} \to \mathit{Spec}_{T,V} \to \mathit{Spec}_{T,U}$$

Ce transformateur est défini inductivement sur la syntaxe comme suit :

$$\begin{array}{lcl}
\mathit{wpr}\ (\textbf{effect}\ f)\ R & \stackrel{\text{def}}{=} & \lambda\ e\ e' \Rightarrow R\ e\ (f\ e') \\
\mathit{wpr}\ (S_1;S_2)\ R & \stackrel{\text{def}}{=} & \mathit{wpr}\ S_1\ (\mathit{wpr}\ S_2\ R) \\
\\
\mathit{wpr}\ (\textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end})\ R & \stackrel{\text{def}}{=} & \lambda\ e\ e' \Rightarrow \begin{array}{r} (C\ e') \wedge (\mathit{wpr}\ S_1\ R\ e\ e') \\ \vee\ \ \neg(C\ e') \wedge (\mathit{wpr}\ S_2\ R\ e\ e') \end{array} \\
\\
\mathit{wpr}\ (\textbf{while}\ C\ \textbf{do}\ S\ \textbf{end})\ R & \stackrel{\text{def}}{=} & \mathit{lfp}\ (\lambda\ \mathcal{X}\ e\ e' \Rightarrow \begin{array}{r} (C\ e') \wedge (\mathit{wpr}\ S\ \mathcal{X}\ e\ e') \\ \vee\ \ \neg(C\ e') \wedge (R\ e\ e')) \end{array} \\
\\
\mathit{wpr}\ \langle R_2 \rangle\ R_1 & \stackrel{\text{def}}{=} & \kappa\ R_2\ R_1
\end{array}$$

Nous allons maintenant donner une description plus détaillée de wpr pour chaque type d'instruction S, en expliquant informellement pourquoi (wpr S R) calcule bien une formule logique équivalente à (κ $[\![S]\!]$ R).

Affectation. Il s'agit ici de calculer une formulation de l'ensemble des comportements (e, e') qui, séquencés avec l'affectation (**effect** f), donnent lieu à un comportement global

admissible par R. Après avoir exécuté $\langle \mathsf{wpr}\ (\textbf{effect}\ f)\ R\rangle$ sur un état initial e, on produit un état e_x tel que $R\ e\ (f\ e_x)$ est vraie. Si ensuite on exécute $(\textbf{effect}\ f)$ sur e_x, on produit un état e' tel que $e' = (f\ e_x)$. Le comportement (e, e') qui résulte de cette séquence est donc tel que $(R\ e\ e')$ est vraie. Par conséquent, il s'agit bien d'un comportement admissible par $\langle R\rangle$.

Séquence. L'intuition sous-jacente à la définition de $(\mathsf{wpr}\ (S_1;S_2)\ R)$ est la suivante. Nous cherchons K_1 tel que $K_1;S_1;S_2 \sqsubseteq R$. Dans un premier temps, on calcule récursivement $K_2 = (\mathsf{wpr}\ S_2\ R)$ tel que $\langle K_2\rangle;S_2 \sqsubseteq R$. Si un tel K_2 n'existe pas, alors il est impossible que $P;S_2$ puisse raffiner R quelque soit P. Mais si un tel K_2 existe, alors il suffit de trouver K_1 tel que $\langle K_1\rangle;S_1$ raffine K_2 pour que, par monotonie de $(;)$, $(K_1;S_1);S_2$ raffine R. Il nous suffit donc de calculer K_1 en exécutant $(\mathsf{wpr}\ S_1\ K_2)$, c'est-à-dire $(\mathsf{wpr}\ S_1\ (\mathsf{wpr}\ S_2\ R))$.

Alternative. Ici, on cherche K tel que $S = (\langle K\rangle;\textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end})$ raffine R. Si nous exécutons S sur un état initial e, on peut distinguer deux cas. Dans le premier cas, $\langle K\rangle$ produit, à partir de e, un état e' tel que $(C\ e')$ est vraie. C'est alors S_1 qui sera ensuite exécuté. Il suffira donc que dans cette situation, $\langle K\rangle$ se comporte de manière à ce que $\langle K\rangle;S_1$ raffine R. Par conséquent, nous n'avons qu'à calculer récursivement un tel K en évaluant $(\mathsf{wpr}\ S_1\ R)$. Dans le second cas, l'état e' produit par $\langle K\rangle$ est tel que $\neg(C\ e')$ est vraie. Et, de manière analogue, nous n'aurons qu'à évaluer $(\mathsf{wpr}\ S_2\ R)$. Ainsi, la définition de wpr pour l'alternative se lit comme suit : soit $(C\ e')$ est vraie et le comportement de K est spécifié par $(\mathsf{wpr}\ S_1\ R)$, ou alors $\neg(C\ e')$ est vraie et le comportement de $\langle K\rangle$ est spécifié par $(\mathsf{wpr}\ S_2\ R)$.

Itération. Soit la fonction F définie ci-dessous :

$$F \stackrel{\text{def}}{=} \lambda\ \mathcal{X} \Rightarrow (\lambda\ e\ e' \Rightarrow (C\ e') \wedge (\mathsf{wpr}\ S\ \mathcal{X}\ e\ e')\ \vee\ \neg(C\ e') \wedge (R\ e\ e'))$$

Compte tenu de la définition de wpr, il est évident que $(\mathsf{wpr}\ (\textbf{while}\ C\ \textbf{do}\ S\ \textbf{end})\ R)$ est un point fixe de F. En effet, on a la dérivation suivante :

$$\begin{array}{ll} & \mathsf{wpr}\ (\textbf{while}\ C\ \textbf{do}\ S\ \textbf{end})\ R \\ \equiv & \mathsf{wpr}\ (\textbf{if}\ C\ \textbf{then}\ S;(\textbf{while}\ C\ \textbf{do}\ S\ \textbf{end})\ \textbf{else}\ [\![\textbf{skip}]\!]\ \textbf{end}))\ R \\ \equiv & \lambda\ e\ e' \Rightarrow (C\ e') \wedge (\mathsf{wpr}\ (S;\textbf{while}\ C\ \textbf{do}\ S\ \textbf{end})\ R\ e\ e')\ \vee\ \neg(C\ e') \wedge (R\ e\ e') \\ \equiv & \lambda\ e\ e' \Rightarrow (C\ e') \wedge (\mathsf{wpr}\ S\ (\mathsf{wpr}\ (\textbf{while}\ C\ \textbf{do}\ S\ \textbf{end})\ R\ e\ e'))\ \vee\ \neg(C\ e') \wedge (R\ e\ e') \\ \equiv & F\ (\mathsf{wpr}\ (\textbf{while}\ C\ \textbf{do}\ S\ \textbf{end})\ R) \end{array}$$

Précédemment, nous avons défini $[\![\textbf{while } C \textbf{ do } S \textbf{ end}]\!]$ comme le plus petit point fixe de

$$G \stackrel{\text{def}}{=} \lambda\, \mathcal{X} \;\Rightarrow\; (\lambda\, e\; e' \Rightarrow (C\; e) \wedge [\![S;\langle \mathcal{X} \rangle]\!]\; e\; e' \;\vee\; \neg(C\; e) \wedge e = e').$$

Par analogie entre les opérateurs (;) dans G, et wpr dans F, la définition de wpr pour les boucles ($\textbf{while } C \textbf{ do } S \textbf{ end}$) est le plus petit point fixe de F. Ce dernier est bien défini puisque, wpr est monotone sur son second argument, donc F aussi est monotone. En conséquence, nous avons les propriétés suivantes.

Lemme 14. *Soit C une condition, et soit S un programme. Soit la fonction F suivante :*

$$F \stackrel{\text{def}}{=} \lambda\, \mathcal{X} \Rightarrow (\lambda\, e\; e' \Rightarrow (C\; e') \wedge (\mathit{wpr}\; S\; \mathcal{X}\; e\; e') \;\vee\; \neg(C\; e') \wedge (R\; e\; e'))$$

Les propriétés suivantes sont valides :

$$(1)\; (\mathit{wpr}\; (\textbf{while } C \textbf{ do } S \textbf{ end})\; R) \;\equiv\; F\; (\mathit{wpr}\; (\textbf{while } C \textbf{ do } S \textbf{ end})\; R)$$
$$(2)\; \forall\, X \cdot ((F\; X) \;\subseteq\; X) \;\rightarrow\; ((\mathit{wpr}\; (\textbf{while } C \textbf{ do } S \textbf{ end})\; R) \;\subseteq\; X)$$

Preuve. D'abord on montre facilement, par induction sur S, que F est monotone. Ensuite, il suffit d'appliquer le théorème du point fixe de Knaster-Tarski. □

Sous-programme. Pour les programmes de la forme $\langle R_2 \rangle$, la plus faible pré-spécification est directement donnée par $(\kappa\; R_2\; R_1)$.

5.3.3 Raffinement et calcul de la plus faible pré-spécification

À présent, nous combinons le transformateur de spécifications wpr avec l'interprétation $[\![\cdot, \cdot]\!]$ pour constituer une condition nécessaire et suffisante de raffinement. Nous commençons par montrer une propriété naturelle de wpr qui est que ce transformateur génère des expressions équivalentes pour les programmes équivalents S et $\langle [\![S]\!] \rangle$. Ceci est formalisé par le lemme suivant.

Lemme 15. $\forall\; S\; R \cdot (\mathit{wpr}\; \langle [\![S]\!] \rangle\; R) \;\equiv\; (\mathit{wpr}\; S\; R)$

Schéma de preuve. On procède par induction sur S. Le cas non trivial est le cas des boucles. Dans la direction ($\rightarrow$), on montre $\forall\, e \cdot (\mathsf{wpr}\; \langle [\![S]\!] \rangle\; R\; e\; e') \rightarrow (\mathsf{wpr}\; S\; R\; e\; e')$ pour tout état e' par induction sur $\prec_S^C$. Et dans la direction ($\leftarrow$), on a une conclusion de la forme $(\mathsf{lfp}\; F) \subseteq X$, on procède donc par induction sur la définition de $(\mathsf{wpr}\; (\textbf{while } C \textbf{ do } S \textbf{ end})\; R)$ (propriété (2) du lemme 14). □

Avec l'aide de ce lemme, nous prouvons aisément le théorème suivant qui permet de définir la notion de raffinement en utilisant le transformateur wpr et l'interprétation $[\![\cdot,\cdot]\!]$.

Théorème 5. $\forall\ S_2\ S_1 \cdot S_2 \sqsubseteq S_1 \quad \leftrightarrow \quad \forall\ e \cdot (\exists\ e' \cdot [\![S_1, [\![\textsf{skip}]\!]]\!]\ e\ e') \rightarrow (\mathit{wpr}\ S_2\ [\![S_1, [\![\textsf{skip}]\!]]\!]\ e\ e)$

Preuve.

$$\forall\ e \cdot (\exists\ e' \cdot [\![S_1, [\![\textsf{skip}]\!]]\!]\ e\ e') \rightarrow (\mathsf{wpr}\ S_2\ [\![S_1, [\![\textsf{skip}]\!]]\!]\ e\ e)$$

$$\leftrightarrow \quad \forall\ e \cdot (\exists\ e' \cdot [\![S_1]\!]\ e\ e') \rightarrow (\mathsf{wpr}\ S_2\ [\![S_1]\!]\ e\ e) \qquad \text{(théorème 4)}$$

$$\leftrightarrow \quad \forall\ e \cdot (\exists\ e' \cdot [\![S_1]\!]\ e\ e') \rightarrow (\mathsf{wpr}\ \langle [\![S_2]\!] \rangle\ [\![S_1]\!]\ e\ e) \qquad \text{(lemme 15)}$$

$$\leftrightarrow \quad \forall\ e \cdot (\exists\ e' \cdot [\![S_1]\!]\ e\ e') \rightarrow (\kappa\ [\![S_2]\!]\ [\![S_1]\!]\ e\ e) \qquad \text{(définition de wpr)}$$

$$\leftrightarrow \quad S_2 \sqsubseteq S_1 \qquad \text{(lemme 13)}$$

□

Maintenant, nous illustrons dans l'exemple ci-dessous, l'application de ce théorème pour prouver un raffinement entre deux programmes qui échangent les valeurs de deux variables.

Exemple 11 (Preuve de raffinement avec wpr)**.** *Soient les programmes S_2 et S_1 suivants.*

$$S_2 \stackrel{\text{def}}{=} x := x - y; y := x + y; x := y - x \qquad S_1 \stackrel{\text{def}}{=} x := x + y; y := x - y; x := x - y$$

Par la dérivation suivante, on montre que $S_2 \sqsubseteq S_1$.

$$S_2 \sqsubseteq S_1$$

$$\leftrightarrow \quad (\exists\ x'\ y' \cdot [\![S_1, [\![\textsf{skip}]\!]]\!]) \rightarrow (\mathit{wpr}\ S_2\ [\![S_1, [\![\textsf{skip}]\!]]\!]) \qquad (\textit{théorème } 5)$$

$$\leftrightarrow \quad \begin{array}{ll} & (\exists\ x'\ y' \cdot x' = x + y - (x + y - y) \wedge y' = x + y - y) \\ \rightarrow & \qquad x + y - (x + y - y) = x - y + y - (x - y) \\ \wedge & \quad x + y - y = x - y + y \end{array} \qquad (\textit{par calcul symbolique})$$

$$\leftrightarrow \quad \mathit{True}$$

Dans l'exemple ci-dessus, l'application du théorème 5 permet ensuite, par calcul symbolique, d'obtenir une formule logique de taille raisonnable dont la validité est facile à établir.

À l'instar de wp, wpr a effectué automatiquement un certain nombre de raisonnements simples. En particulier, dans la formule obtenue par calcul symbolique il n'y a aucun quantificateur existentiel à droite de l'implication. De manière générale, (wpr $(\langle S_1\rangle;...;\langle S_n\rangle)$ R) est de taille linéaire en n et ne contient pas l'opérateur $\Box$, alors que $[\![\langle S_1\rangle;...;\langle S_n\rangle, [\![\textbf{skip}]\!]]\!]$ est de taille exponentielle en n après expansion de l'opérateur $\Box$.

5.4 Le cas des boucles

Les méthodes de preuves que nous avons formalisées jusqu'à présent ne sont pas faciles d'utilisation pour prouver les raffinement de boucles. En effet, elles nécessitent de travailler avec l'opérateur de point fixe du second ordre lfp. Or, comme nous l'avons déjà expliqué dans la section 5.1, il est préférable de travailler en logique du premier ordre. Dans cette section nous étudions les conditions dans lesquelles on peut s'assurer qu'une spécification donnée correspond effectivement à la sémantique d'une boucle, et constitue donc une abréviation qui permet de caractériser cette dernière. Ensuite, nous en déduisons une méthode de preuve de raffinement correcte et complète, permettant de travailler en logique du premier ordre lorsqu'on peut trouver une abréviation du premier ordre.

5.4.1 Abréviation de boucle

Dans [30] Frappier et al. ont dégagé des conditions suffisantes qui permettent de réduire la sémantique d'une boucle à une spécification relationnelle du premier ordre lorsque le corps de la boucle est du premier ordre. Les raisonnements sur de tels boucles peuvent alors être effectués en logique du premier ordre grâce auxdites spécifications qui en sont les sémantiques exactes du premier ordre. Nous inspirant des travaux précités, nous présentons dans le lemme ci-dessous des conditions plus faibles permettant de remplir les mêmes objectifs.

Lemme 16 (Abréviation de boucle)**.** *Soient C une condition et S un programme. Alors, on a :*

$$\begin{array}{lll} \textit{wfd}\ (\lambda\ e\ e' \Rightarrow C\ e' \wedge [\![S]\!]\ e'\ e \wedge C\ e) & & (a)\\ \rightarrow\ \ (\textbf{if}\ C\ \textbf{then}\ S\ \textbf{end; if}\ C\ \textbf{then}\ S\ \textbf{end}) \sqsubseteq \textbf{if}\ C\ \textbf{then}\ S\ \textbf{end} & & (b)\\ \rightarrow\ \ \textbf{while}\ C\ \textbf{do}\ S\ \textbf{end} \equiv \textbf{if}\ C\ \textbf{then}\ \langle\ \lambda\ e\ e' \Rightarrow [\![S]\!]\ e\ e' \wedge \neg(C\ e')\ \rangle\ \textbf{end} & & \end{array}$$

Schéma de preuve. On suppose (a) et (b). Ensuite on doit établir que

$$[\![\textbf{while}\ C\ \textbf{do}\ S\ \textbf{end}]\!] \equiv [\![\textbf{if}\ C\ \textbf{then}\ \langle\ \lambda\ e\ e' \Rightarrow [\![S]\!]\ e\ e' \wedge \neg(C\ e')\ \rangle\ \textbf{end}]\!]$$

Dans la direction ($\subseteq$), il suffit de procéder par induction de point fixe en appliquant le lemme 6 (propriété (2)). Dans l'autre direction, il suffit d'appliquer le principe d'induction associé à l'hypothèse (a) et d'utiliser l'hypothèse (b). □

De manière informelle, la condition (a) du lemme ci-dessus permet de garantir que la boucle termine pour tous les états initiaux appartenant au domaine de la spécification $R \stackrel{\text{def}}{=} \lambda\ e\ e' \Rightarrow (C\ e) \wedge [\![S]\!]\ e\ e'$. Quand à la condition ($b$) du lemme, elle conduit à la conséquence que les comportements admissibles par deux tours de boucles successifs sont aussi admissibles par un seul tour de boucle, soit par $W = (\textsf{if}\ C\ \textsf{then}\ S\ \textsf{end})$. Donc, par monotonie, un seul tour de boucle contient tous les comportements admissibles de la boucle. Sachant que la boucle termine nécessairement sur un état e' tel que $\neg(C\ e')$ est vrai, il suffit donc de restreindre le codomaine de W à l'ensemble des états e' tels que $\neg(C\ e')$ est vrai pour obtenir une caractérisation de tous les comportements admissibles de la boucle. Une telle restriction de codomaine correspond exactement à l'abréviation suivante $A_w \stackrel{\text{def}}{=} (\textsf{if}\ C\ \textsf{then}\ \langle\ \lambda\ e\ e' \Rightarrow [\![S]\!]\ e\ e' \wedge \neg(C\ e')\ \rangle\ \textsf{end})$. Dans l'exemple suivant, nous illustrons l'utilisation du lemme 16 pour établir la validité d'une abréviation de boucle.

Exemple 12. *Considérons l'assertion suivante d'abréviation d'une boucle :*

$$\textsf{while}\ i \neq n\ \textsf{do}\ \langle i < i' \leq n \rangle_n\ \textsf{end} \quad \equiv \quad \textsf{if}\ i \neq n\ \textsf{then}\ \langle i < i' \leq n \wedge i' = n' \rangle_n\ \textsf{end}$$

Cette équivalence peut être prouvée en appliquant le lemme 16. En effet la condition (a) du lemme est remplie puisque la spécification ($i \neq n \wedge i < i' <= n \wedge i' \neq n' \wedge n = n'$) augmente strictement et systématiquement la valeur entière de i, sans que cette dernière ne puisse dépasser la valeur de n qui elle , est maintenue constante. De plus la condition (b) aussi est remplie. Pour établir ce fait, il suffit d'appliquer le théorème 5. On verra alors que la condition (b) se simplifie en la condition suivante :

$$\begin{array}{l} i < i' <= n \wedge i' \neq n' \wedge n = n' \ \wedge\ i' < i'' <= n' \wedge i'' \neq n'' \wedge n' = n'' \\ \rightarrow \quad i < i'' <= n \wedge i'' \neq n'' \wedge n = n'' \end{array}$$

Comme cette condition est valide, on a bien la validité de l'abréviation considérée.

5.4.2 Méthode de preuve de raffinement pour boucles

La technique d'abréviation des boucles présentée dans la section précédente nous donne une méthode de preuve de raffinement de boucle. En effet, considérons un programme R, et la boucle $B \stackrel{\text{def}}{=} (\textsf{while}\ C\ \textsf{do}\ P\ \textsf{end})$. Si nous arrivons à trouver une abréviation A_B du premier ordre pour B, alors pour prouver que B raffine R, il nous suffit d'appliquer

le lemme d'abréviation (lemme 16), et de montrer ensuite que A_B raffine R. Ce faisant, nous nous aurons épargné de devoir travailler avec la sémantique de point fixe de la boucle qui est une sémantique d'ordre supérieur. Nous illustrons rapidement cette méthode dans l'exemple ci-dessous. Puis nous montrerons que cette méthode est correcte, et de plus complète relativement à l'expressivité du langage de spécification.

Exemple 13. *Considérons le raffinement suivant :*

$$\textbf{while } i \neq n \textbf{ do } i := i + 1 \textbf{ end } \sqsubseteq \langle i \leq i' \leq n \wedge i' = n' \rangle_n$$

Pour montrer la validité de ce raffinement, nous procédons en deux étapes. Premièrement, nous pouvons aisément établir que $(i := i + 1)$ *raffine* $S \stackrel{\text{def}}{=} \langle i < i' \leq n \rangle$*. Et donc, nous avons par monotonie que*

$$\textbf{while } i \neq n \textbf{ do } i := i + 1 \textbf{ end } \sqsubseteq \textbf{ while } i \neq n \textbf{ do } S \textbf{ end}$$

Deuxièmement, nous avons montré dans l'exemple 12 que la boucle **while** $i \neq n$ **do** S **end** *était équivalente à l'abréviation* $A \stackrel{\text{def}}{=}$ **if** $i \neq n$ **then** $\langle i < i' \leq n \wedge i' = n' \rangle_n$ **end**. *Par conséquent, la validité du raffinement considéré se déduit par transitivité, de la validité du raffinement suivant :*

$$A \sqsubseteq \langle i \leq i' \leq n \wedge i' = n' \rangle_n$$

Contrairement au raffinement initialement considéré, le raffinement ci-dessus peut être prouvé en logique du premier ordre par application des méthodes précédemment établies.

Maintenant, nous montrons que la méthode de preuve que nous venons d'esquisser est correcte et relativement complète. Ceci est formellement énoncé dans le théorème ci dessous. Ce théorème indique que pour prouver un raffinement de boucle, il faut et il suffit de trois conditions. La première est de trouver un programme S qui remplit les conditions d'abréviation du lemme 16, ici il s'agit des conditions (a) et (c). La deuxième condition est que le corps de la boucle P raffine S. Enfin, la dernière condition est que l'abréviation de la boucle raffine effectivement la spécification R.

Théorème 6 (Conditions nécessaires et suffisantes de raffinement de boucle)**.** *Soit C une condition, et soient P et R deux programmes. Alors, on a :*

$$\textbf{while } C \textbf{ do } P \textbf{ end } \sqsubseteq R$$

si et seulement si

$$\exists\, S \cdot \begin{array}{ll} \mathit{wfd}\ (\lambda\ e\ e' \Rightarrow C\ e' \wedge [\![S]\!]\ e'\ e \wedge C\ e) & (a) \\ \wedge\ P \sqsubseteq S & (b) \\ \wedge\ (\textbf{if } C \textbf{ then } S \textbf{ end});(\textbf{if } C \textbf{ then } S \textbf{ end}) \sqsubseteq (\textbf{if } C \textbf{ then } S \textbf{ end}) & (c) \\ \wedge\ \textbf{if } C \textbf{ then } \langle\ \lambda\ e\ e' \Rightarrow [\![S]\!]\ e\ e' \ \wedge\ \neg(C\ e')\ \rangle \textbf{ end } \sqsubseteq R & (d) \end{array}$$

Schéma de preuve. Dans la direction ($\rightarrow$) on prend $S = \langle\ (\prec_P^C)^+\ \rangle$, choisissant ainsi la fermeture transitive de $\prec_P^C$ comme témoin. Nous devons alors établir la validité de (a), (b), (c) et (d). Pour prouver la validité de (a), on observe que $\prec_P^C$ est une relation bien fondée (théorème 1), et donc S aussi est bien fondée puisque la fermeture transitive d'une relation bien fondée l'est également. On observe ensuite que la relation $[\![S]\!]$ contient la relation $(\lambda\ e\ e' \Rightarrow C\ e' \wedge [\![S]\!]\ e'\ e \wedge C\ e)$. Par conséquent cette dernière est bien fondée puisque toute relation contenue dans une relation bien fondée l'est également. Pour établir la condition (b) commençons par constater que toute relation raffine sa fermeture transitive. En effet, la fermeture transitive d'une relation préserve le domaine de la relation mais est plus non-déterministe que cette dernière. Nous avons donc que $\langle\ \prec_P^C\ \rangle$ raffine S. On peut de plus constater que P raffine $\langle\ \prec_P^C\ \rangle$ car cette dernière ne fait que restreindre le domaine de P. Ces deux constats nous permettent de déduire par transitivité que P raffine S. En ce qui concerne les conditions (c) et (d) il suffit de procéder par induction de point fixe sur la sémantique de (**while** C **do** P **end**) pour (c), et par induction bien fondée sur $(\prec_P^C)$ pour (d).

Dans la direction ($\leftarrow$), on a **while** C **do** P **end** $\sqsubseteq$ **while** C **do** S **end** par monotonie du raffinement et par l'hypothèse (b). Par (a), (b), (c), et par le lemme 16, on a

$$\textbf{while } C \textbf{ do } S \textbf{ end } \equiv \textbf{ if } C \textbf{ then } \langle\ \lambda\ e\ e' \Rightarrow [\![S]\!]\ e\ e' \ \wedge\ \neg(C\ e')\ \rangle \textbf{ end}$$

On peut donc, par l'hypothèse (d), conclure que **while** C **do** P **end** $\sqsubseteq R$. □

À présent, nous illustrons, dans l'exemple suivant, l'utilisation de ce théorème pour établir un raffinement de boucle.

Exemple 14 (Recherche linéaire)**.** *Considérons le programme suivant.*

$$
\begin{array}{rcl}
Q & \stackrel{\text{def}}{=} & \textsf{while } i \neq n \textsf{ do} \\
 & & \quad \textsf{if } a[i] = x \textsf{ then } n := i \textsf{ else } i := i + 1 \textsf{ end} \\
 & & \textsf{end}
\end{array}
$$

Le programme Q recherche la valeur de x dans le tableau a. On suppose que a est de taille supérieure ou égale à n, et que la recherche concerne le segment du tableau qui commence à la valeur initiale de i et se termine à l'indice $(n-1)$ du tableau. Considérons aussi le programme abstrait P suivant.

$$P \stackrel{\text{def}}{=} \langle\ i \leq n \wedge ((\forall\ k \cdot i \leq k < n \rightarrow a[k] \neq x)\ \vee\ i' < n \wedge a[i'] = x)\ \rangle_{a,x}$$

Le programme P est caractérisé par la spécification suivante. Lorsque P termine, il y a deux possibilités : (1) soit $x \notin a[i..n[$, (2) soit la variable i contient un indice du tableau où on peut trouver x. De plus, le contenu de la variable x et de la variable a à la fin de l'exécution ne doit pas différer du contenu de ces variables en début d'exécution. En utilisant le théorème 6 nous pouvons établir que $Q \sqsubseteq P$. Pour commencer, on choisit une abstraction S du corps de la boucle définie comme suit :

$$S \stackrel{\text{def}}{=} \langle\ i < i' \leq n \wedge (\forall\ k \cdot i \leq k < i' \rightarrow a[k] \neq x) \wedge n = n'\ \vee\ i' < n \wedge a[i'] = x \wedge i' = n'\ \rangle_{a,x}$$

Cette abstraction exprime deux comportements possibles du corps de la boucle. Le premier comportement consiste à déplacer le curseur i de la recherche vers n sans pour autant dépasser cette valeur $(i < i' \leq n)$, tout en nous assurant que nous n'avons pas manqué la valeur recherchée $(\forall\ k \cdot i \leq k < i' \rightarrow a[k] \neq x)$. Le second comportement correspond au cas où on connaît un indice où trouver x dans a. Dans ce cas, nous serons en mesure de faire en sorte que i contienne un indice où trouver x. Ceci revient à se conformer au comportement spécifié par $\langle a[i'] = x \rangle$. Pour ce faire, nous pourrons, par exemple, copier cet indice connu dans la variable i. Une fois que nous avons trouvé la valeur recherchée, nous pouvons arrêter l'exécution en invalidant la condition de la boucle, c'est-à-dire en nous conformant au comportement spécifié par $\langle i' = n' \rangle$. En un sens, il s'est agit pour nous de spécifier le comportement de la boucle de la manière la plus abstraite possible pour nous réserver la plus grande marge de manœuvre possible en termes d'implantation. Muni de S, nous devons maintenant démontrer que les conditions pour appliquer le théorème 6 sont remplies. Ces preuves sont triviales mais assez fastidieuses sur papier, donc nous ne les détaillerons pas ici.

Notons que la condition (b) du lemme d'abréviation des boucles (lemme 16) implique que la relation $(\lambda\ e\ e' \Rightarrow C\ e \wedge [\![S]\!]\ e\ e')$ est transitive. Ce qui signifie aussi que dans le cas général, cette relation doit être non-déterministe pour que l'abréviation de boucle soit applicable. Ainsi, la capacité de spécifier des comportements non-déterministes est incontournable, y compris lorsque le programme final visé est déterministe. On notera aussi que les résultats de cette section ont été établis en regard de la sémantique brute $[\![\cdot]\!]$, cependant ils se transfèrent aisément à la sémantique alternative $[\![\cdot, [\![\textsf{skip}]\!]]\!]$ par le biais du théorème 4. Dans la pratique, nous utiliserons plutôt les résultats transférés afin d'automatiser un maximum de raisonnements.

5.5 Conclusion

Dans ce chapitre, nous avons formalisé des résultats permettant d'établir la validité d'un raffinement donné. À l'instar du wp-calcul, ces résultats permettent d'automatiser certains raisonnements fastidieux comme l'élimination des quantificateurs existentiels dans les cas triviaux. Afin de rendre notre formalisation aussi générale que possible, nous avons établi ces résultats autour du concept de plus faible pré-spécification au lieu de celui de plus faible précondition, le premier concept étant une généralisation du second.

Pour contourner les difficultés que peuvent poser le raisonnement sur les boucles basé sur l'opérateur du second ordre lfp, nous avons formalisé une technique d'abréviation des boucles en logique du premier ordre inspiré des travaux de Frappier et al. [30]. Cependant, à la différence de ces travaux, notre technique d'abréviation requiert des conditions plus faibles et mène à une méthode de preuve de raffinement de boucles qui est prouvée relativement complète. Une autre technique d'abréviation de boucles proposée par Sekerinski [70] a été prouvée relativement complète par Tchier [75]. Cependant, cette dernière requiert des conditions plus fortes que celles que nous avons dégagées. En effet, dans notre formalisation, utiliser ces conditions alternatives reviendrait à remplacer, dans notre condition (b) du lemme d'abréviation, la relation de raffinement par une relation plus forte d'équivalence. De plus, notre condition (a) de bonne fondation du lemme d'abréviation est plus faible que la condition correspondante chez Sekerinski, Frappier, et Tchier puisque notre condition de bonne fondation concerne une relation plus réduite.

Chapitre 6

Programmation par raffinements

Dans ce chapitre nous nous intéressons à une approche à la fois progressive et modulaire de la programmation ainsi que de la preuve de programmes. Il s'agit de l'approche par raffinements successifs initiée par Wirth [80] et Dijkstra [24]. Nous commençons par quelques éléments de motivation dans la section 6.1. Ensuite, dans la section 6.2, nous illustrons par un exemple le processus de développement par raffinements dans un formalisme prédicatif et relationnel. Dans la section 6.3, nous étendons notre langage de programmation afin d'en faire un langage capable de décrire l'articulation des raffinements successifs qui constituent un développement donné. Enfin, dans la section 6.4 nous formulons une logique de raisonnement sur les développements par raffinements, puis nous discutons la correction ainsi que la complétude de cette logique.

6.1 Motivation

Les programmes informatiques sont comme beaucoup de textes plus souvent lus qu'ils ne sont écrits ou modifiés. Mais bien souvent, à la différence des textes littéraires, la lecture des instructions seules ne permet pas de comprendre pourquoi un programme fonctionne, mais uniquement ce que le programme fait. D'où l'importance des commentaires qui en complément permettent d'expliquer en quoi ce que fait le programme est adéquat. Cependant, même dans un code bien commenté, la tenue à jour des commentaires lorsque le programme évolue reste un défi d'autant plus que ni l'absence de commentaires, ni leur inadéquation n'affectent en rien la compilation ou l'exécution des programmes. La difficulté de compréhension d'un programme est une des causes d'introduction d'erreur lors de la maintenance des programmes sachant que cette période de maintenance représente en général la majeure partie de la durée de vie d'un programme. Pour être véritablement utiles, les commentaires doivent dépasser la simple paraphrase du code et plutôt expliquer

les objectifs de plus haut niveau visés par telle ou telle partie du programme. C'est ce que permet, par exemple, le style de la programmation dite *lettrée* [42] dont nous avons emprunté la notation $\langle ... \rangle$. Dans cette approche de la programmation, le code est organisé en fragments de programmes commentés. Idéalement, il s'agit en fait pour le programmeur de faire état du processus de réflexion ayant finalement mené à la solution qui a été implémentée.

À ce besoin d'intelligibilité des programmes s'ajoute un besoin de modularité dans la conception et la vérification des programmes. D'où l'approche par étapes successives de raffinements [80, 24] qui combinées ensemble permettent de mener à un résultat nécessairement correct lorsque toutes les étapes le sont. Comme l'a montré Morgan [56], cette approche revient aussi à organiser le code en un assemblage de fragments dont les objectifs sont décrits précisément par des spécifications. Pratiqué dans un cadre formel, cette approche conduit à des programmes plus intelligibles et permet d'élever le niveau de confiance dans la mesure où toutes les étapes de raffinements sont prouvées corrects. Quelque part, l'artefact de référence n'est plus le code source du programme exempt d'erreurs de compilation, mais plutôt un objet sémantiquement plus riche qui ne se contente pas seulement de dire ce que la machine doit faire (instructions de programmes), mais qui en plus fait état de ce que l'on veut que la machine fasse (spécifications). Comme Morgan dans [58], nous désignerons un tel objet par le terme *développement*. Une fois un développement achevé et prouvé correct il est alors relativement aisé d'en extraire, par un mécanisme de traduction (ou d'*effacement* des éléments non calculatoires), un programme dans un langage pour lequel on dispose d'un compilateur.

Ces besoins d'intelligibilité et de modularité ont beaucoup été étudiés dans le cadre de la logique de Hoare, du calcul de la plus faible précondition, ou encore dans celui du calcul des relations. Il est donc naturel que nous l'étudions aussi dans le cadre prédicatif, relationnel et typé que nous avons posé dans les chapitre précédents. De plus, il y a la perspective d'un niveau de confiance plus élevé grâce à la possibilité de produire un certificat de correction sous la forme d'un λ-terme dont la vérification peut s'affranchir des méthodes et des environnements de développements mis en œuvre.

6.2 Un exemple de développement par raffinements

Le processus de développement par raffinements commence par l'expression d'une spécification de haut niveau du problème à résoudre. Prenons comme exemple de problème le calcul de la racine carrée d'un entier positif. D'abord nous donnerons une spécification formelle à ce problème, et ensuite développons une solution en quatre étapes de raffinements.

Spécification. Une possible spécification informelle du problème est la suivante. Soit une variable x de type entier positif, concevoir un algorithme qui calcule la racine carrée de x et qui place le résultat dans une autre variable r de même type. De plus, on souhaite que la valeur de l'entrée x reste inchangée. Autrement dit, à la fin de l'exécution de l'algorithme nous devons pouvoir constater deux choses. Premièrement, la valeur finale r' de la variable r doit avoir la propriété d'être la racine entière de x. Et deuxièmement, la valeur initiale de la variable x doit être égale à sa valeur finale x'. En utilisant la construction $\langle R \rangle$ de notre langage, nous pouvons formellement traduire notre spécification de départ comme suit :

$$\mathsf{Sqrt} := \langle \overbrace{r'^2 \leq x < (r'+1)^2}^{r'=\sqrt{x}} \ \wedge \ x = x' \rangle \tag{6.1}$$

Pour le moment, nous ne savons pas encore exécuter l'algorithme ainsi spécifié car la spécification est trop abstraite. Cependant, nous pouvons aisément nous convaincre (ou convaincre autrui) que l'on a une spécification correcte du comportement observable de l'algorithme visé. Ici, la construction $\langle R \rangle$ est à interpréter comme une *instruction de spécification* [57] représentant un fragment de programme dont l'implémentation concrète reste à préciser. Naturellement, l'étape suivante va consister à *développer* cette spécification pour la rendre plus précise, faisant ainsi un premier pas vers un algorithme exécutable.

Premier raffinement. Pour cette première étape de raffinement nous nous intéressons à l'initialisation de l'algorithme, soit en l'occurrence à la valeur initiale à affecter à la variable r. Pour ce faire nous partons du constat suivant :

$$0 \leq \sqrt{x} < (x+1)$$

D'où nous tirons que le résultat recherché r' est tel que :

$$0 \leq r'^2 \leq x < (r'+1)^2 \leq (x+1)^2 \tag{6.2}$$

Nous connaissons donc les bornes d'un intervalle dans lequel se situe le résultat recherché. Nous pouvons donc initialiser r à 0 et ensuite chercher la solution en augmentant r progressivement, ou nous pouvons initialiser r à $(x+1)$ et ensuite chercher la solution en diminuant r progressivement. Néanmoins, plutôt que de faire ce choix maintenant, nous préférons différer ce dernier à plus tard. Pour ce faire, nous nous donnons un degré de

liberté supplémentaire en introduisant une variable locale h. Nous pouvons alors initialiser r à 0 et h à $(x+1)$ ce qui nous donne deux possibilités de progresser vers la solution, soit en augmentant r, soit en diminuant h. Après avoir ainsi initialisé r et h, il est facile de voir que le problème restant à résoudre peut s'écrire comme suit :

$$\langle\ r^2 \leq r'^2 \leq x < (r'+1)^2 \leq h^2\ \rangle_x \tag{6.3}$$

En effet, en résolvant ce dernier on se ramène au problème 6.2 puisque :

$$r := 0; h := x+1; \langle\ r^2 \leq r'^2 \leq x < (r'+1)^2 \leq h^2\ \rangle_x \ \equiv\ \langle\ 0 \leq r'^2 \leq x < (r'+1)^2 \leq (x+1)^2\ \rangle_x$$

Afin de capturer notre première étape de raffinement, nous utilisons la notion de bloc spécifié [32] qui associe une spécification abstraite appelée *abstraction*, et une implémentation plus concrète appelée *concrétisation*. Par conséquent, comme illustré ci-dessous, nous modifions la définition 6.1 en insérant entre accolades la résultat du raffinement.

$$\begin{array}{l}\mathsf{Sqrt} := \langle\ r'^2 \leq x < (r'+1)^2\ \rangle_x\ \{ \\ \qquad r := 0; \\ \qquad h := x+1; \\ \qquad \langle\ r^2 \leq r'^2 \leq x < (r'+1)^2 \leq h^2\ \rangle_x \\ \}\end{array}$$

Notre développement devient alors un bloc spécifié dont l'abstraction est la spécification initiale, et dont la concrétisation consiste en une initialisation de variables suivie d'une spécification abstraite à préciser plus tard. Il convient bien entendu de prouver formellement que ce raffinement est correct, et pour ce faire nous pouvons appliquer les techniques décrites dans le chapitre précédent. Maintenant, passons au raffinement suivant qui va consister à préciser une implémentation pour le problème (6.3).

Deuxième raffinement Un examen de la relation (6.3) permet de constater que si $h = r + 1$ alors l'implémentation triviale **skip** suffit puisque on peut aisément montrer que :

$$\textbf{skip} \ \sqsubseteq\ \langle\ r^2 \leq r'^2 \leq x < (r'+1)^2 \leq (r+1)^2\ \rangle_x$$

Il semble donc raisonnable de chercher à se ramener à la situation dans laquelle $h = r+1$. Pour se ramener à cette situation, on peut par exemple adopter une stratégie itérative consistant en deux étapes. La première étape est de généraliser le problème (6.3) en remplaçant l'expression $(r + 1)$ par la variable h, ce qui donne la spécification suivante

d'un problème plus général :

$$\langle\ r^2 \leq r'^2 \leq x < h'^2 \leq h^2\ \rangle_x \tag{6.4}$$

Et la deuxième étape consiste à itérer le problème généralisé jusqu'à ce que la condition $h = r+1$ soit vraie. Notre raisonnement nous a ainsi mené à introduire une boucle dont la condition d'arrêt est $h = r+1$ et dont le corps est spécifié par la relation (6.4). Cependant, cette dernière spécification ne tient pas encore compte de la nécessité pour la boucle de s'arrêter. Nous devons donc la compléter avec une exigence de progression. Dans notre cas il suffit de préciser que le corps de la boucle doit modifier au moins une des variables r et h. Après ce deuxième raffinement, notre développement s'enrichit à nouveau comme illustré ci-dessous.

$\mathsf{Sqrt} := \langle\ r'^2 \leq x < (r'+1)^2\ \rangle_x\ \{$
$\quad r := 0;$
$\quad h := x + 1;$
$\quad \langle\ r^2 \leq r'^2 \leq x < (r'+1)^2 \leq h^2\ \rangle_x\ \ \{$
$\quad\quad$ **while** $h \neq r + 1$ **do**
$\quad\quad\quad \langle\ r^2 \leq r'^2 \leq x < h'^2 \leq h^2\ \wedge\ \underbrace{(r', h') \neq (r, h)}_{\text{progression}}\ \rangle_x$
$\quad\quad$ **end**
$\quad \}$
$\}$

Troisième raffinement. Nous focalisons désormais nos efforts sur le raffinement du corps de la boucle. Ici, nous devons modifier r, ou h, ou les deux de sorte à remplir la spécification suivante : $\langle\ r^2 \leq r'^2 \leq x < h'^2 \leq h^2\ \rangle$. On remarque que pour tout m pris dans $]r, h[$ on a trois cas. Dans le premier cas, on a $m^2 < x$ et on a clairement trouvé une nouvelle valeur pour r puisque m est telle que :

$$r^2 \leq m^2 \leq x < h'^2 \leq h^2\ \wedge\ (m, h') \neq (r, h)$$

Dans le second cas, on a $m^2 > x$ et on a trouvé une nouvelle valeur pour h puisque m est telle que :

$$r^2 \leq r'^2 \leq x < m^2 \leq h^2\ \wedge\ (r', m) \neq (r, h)$$

Enfin dans le troisième cas, on a $m^2 = x$ donc on a trouvé une nouvelle valeur à r et à h puisque :

$$r^2 \leq m^2 \leq x < (m+1)^2 \leq h^2\ \wedge\ (m, m+1) \neq (r, h)$$

Notre troisième raffinement va donc consister d'abord à sélectionner un pivot dans l'intervalle de recherche, et ensuite à affecter à r et à h leur nouvelle valeur en fonction des trois cas que nous venons de distinguer. L'état de notre développement est alors indiqué ci-dessous.

```
Sqrt := ⟨ r'^2 ≤ x < (r'+1)^2 ⟩_x {
        r := 0;
        h := x + 1;
        ⟨ r^2 ≤ r'^2 ≤ x < (r'+1)^2 ≤ h^2 ⟩_x  {
            while h ≠ r + 1 do
                ⟨ r^2 ≤ r'^2 ≤ x < h'^2 ≤ h^2  ∧  (r', h') ≠ (r, h) ⟩_x {
                    ⟨ r < m' < h ⟩_{x,r,h};
                    if m^2 < x then r := m
                    else if m^2 > x then h := m
                    else r := m; h := m + 1 end
                }
            end
        }
}
```

Quatrième raffinement. Pour terminer notre développement, il nous reste à préciser plus concrètement le choix du pivot m. Le développement final est celui de la figure 6.2 ci-dessous.

```
Sqrt := ⟨ r'^2 ≤ x < (r'+1)^2 ⟩_x {
        r := 0;
        h := x + 1;
        ⟨ r^2 ≤ r'^2 ≤ x < (r'+1)^2 ≤ h^2 ⟩_x  {
            while h ≠ r + 1 do
                ⟨ r^2 ≤ r'^2 ≤ x < h'^2 ≤ h^2  ∧  (r', h') ≠ (r, h) ⟩_x {
                    ⟨ r < m' < h ⟩_{x,r,h} { m := r + (h − r)/2 };
                    if m^2 < x then r := m
                    else if m^2 > x then h := m
                    else r := m; h := m + 1 end
                }
            end
        }
}
```

Figure 6.1: Développement final de l'algorithme Sqrt

Nous pourrions choisir comme implémentation $m := r + 1$ (respectivement $m := h - 1$)

ce qui correspondrait à une recherche linéaire à partir de la borne inférieure (respectivement supérieure) de l'intervalle. Pour des raisons de performance nous choisirons le pivot au milieu de l'intervalle, ainsi procédant par dichotomie et conférant à l'algorithme une complexité logarithmique. Notons que le processus de raffinement permet de confiner ces considérations à cette ultime étape de raffinement. L'utilité de ce confinement réside dans le fait que dans le reste de la preuve de l'algorithme on a pu complètement s'abstraire des détails qui entre en considération à l'occasion de ce dernier raffinement. Nous sommes arrivé au terme du processus de raffinement car nous avons donné une implémentation concrète à toutes les spécifications abstraites. Bien que l'objet final obtenu n'est pas un programme que l'on peut directement compiler et exécuter, il est facile d'en extraire un programme. En effet, il suffit d'ignorer les spécifications intermédiaires. L'intérêt d'avoir effectué ce développement sous formes de blocs imbriqués hiérarchiquement est que la modification du corps d'un bloc voit son impact limité à la preuve de correction du bloc. Par conséquent, la modification d'un bloc a un impact local sur la preuve du programme plutôt que d'avoir un impact global. De plus, tous les niveaux d'abstractions sont présents dans le développement final. On a donc la possibilité de cacher certains blocs pour avoir une vue abstraite du développement, ou au contraire d'expanser d'autres blocs pour examiner les détails de l'implémentation.

6.3 Langage de développement

Le langage de programmation que nous avons défini précédemment contient la plupart des structures nécessaires à la description du développement présenté dans la figure 6.2, à l'exception des blocs spécifiés. La syntaxe du langage correspondant à ce développement est donc simplement l'extension de notre langage de programmation avec une structure supplémentaire de bloc spécifié. Cette syntaxe est présentée dans la figure 6.2 ci-dessous.

Un développement D est paramétré par les types T et T', ainsi que par un booléen b. Comme dans le cas de notre langage de programmation, les types T et T' représentent des états de programme. Le booléen b permet de distinguer les développements qui contiennent des blocs spécifiés (quand $b = \mathsf{false}$), des développements qui n'en contiennent pas (quand $b = \mathsf{true}$). Par exemple, la construction de bloc spécifié a pour type $\mathsf{Dev}^{T,T,\mathsf{false}}$, de même que tous les développements qui contiennent des blocs directement ou indirectement. La nécessité de cette distinction a à voir avec la construction de bloc spécifié. En effet, en

$\mathsf{Dev}\ D^{T,T',\mathsf{true}} ::=$
$\quad\mid$ **effect** f (transformateur d'état avec $f : T \to T'$)
$\quad\mid$ $\langle R \rangle$ (instruction de spécification avec $R : \mathsf{Spec}_{T,T'}$)

$\mathsf{Dev}\ D^{T,T',(b_1\ \&\&\ b_2)} ::=$
$\quad\mid$ D_1^{T,U,b_1} **;** D_2^{U,T',b_2} (séquence)
$\quad\mid$ **if** C **then** D_1^{T,T',b_1} **else** D_2^{T,T',b_2} **end** (alternative avec la condition $C : T \to \mathsf{Prop}$)

$\mathsf{Dev}\ D^{T,T,b} ::=$
$\quad\mid$ **while** C **do** $D^{T,T,b}$ **end** (itération avec la condition $C : T \to \mathsf{Prop}$)

$\mathsf{Dev}\ D^{T,T',\mathsf{false}} ::=$
$\quad\mid$ $D_1^{T,T',\mathsf{true}}$ { $D_2^{T,T',b}$ } (bloc spécifié)

Figure 6.2: Langage de développement

l'absence du paramètre b, le développement suivant serait autorisé par la syntaxe.

$$(\langle\ S\ \rangle\ \{\ P\ \})\ \ \{\ D\ \}$$

Ce développement n'a pas véritablement de sens car, l'abstraction du bloc étant lui même un bloc, il communique une intention de raffiner cette abstraction, donc de raffiner un bloc. Or, l'interprétation d'un bloc comme une spécification ne semble pas très naturelle.

Afin de restreindre la syntaxe des développements et éviter que l'on puisse utiliser des blocs dans les abstractions de blocs, nous utilisons le paramètre b. En particulier, le type des abstractions de bloc est $\mathsf{Dev}^{T,T,\mathsf{true}}$. Ce type correspond à celui des développements qui ne contiennent pas de blocs et qui peuvent donc s'interpréter comme des programmes, qui s'interprètent aisément comme des spécifications. Notons que les expressions de type $\mathsf{Dev}^{T,T',\mathsf{true}}$ peuvent utiliser les structures de contrôle. Dans certains cas, ceci permet d'améliorer la lisibilité. Par exemple, une abstraction de bloc peut être de la forme if -then -else. Plus généralement, les types $\mathsf{Dev}^{T,T',\mathsf{true}}$ et $\mathsf{Stmt}^{T,T'}$ sont isomorphes comme le montre le lemme suivant :. Le fait que $\mathsf{Dev}^{T,T',\mathsf{true}}$ et $\mathsf{Stmt}^{T,T'}$ sont isomorphes nous permet d'assimiler les deux types et de transférer tous nos résultats des chapitres précédents.

Lemme 17. *Les types $\mathsf{Dev}^{T,T',\mathsf{true}}$ et $\mathsf{Stmt}^{T,T'}$ sont isomorphes.*

Preuve. On définit aisément une transformation $\mathsf{s2d} : \mathsf{Stmt}^{T,T'} \to \mathsf{Dev}^{T,T',\mathsf{true}}$. Dans les cas de base, cette transformation envoie respectivement les instructions (**effect** f) et $\langle\ R\ \rangle$ sur les développements (**effect** f) et $\langle\ R\ \rangle$. Dans les autres cas elle procède récursivement en envoyant, par exemple, **while** C **do** S **end** sur le développement **while** C **do** (s2d S) **end**.

Il suffit ensuite de montrer que la transformation s2d est une bijection. □

Dans le cas de la séquence ou de l'alternative où les développements sont construits à partir de deux développements de types respectifs Dev^{T,T',b_1} et Dev^{T,T',b_2}, le paramètre b est la conjonction de b_1 et b_2 puisque le développement global ne contient pas de blocs si et seulement si aucun de ses constituants n'en contient.

6.4 Correction des développements

Dans cette section, nous nous intéressons à la question du raisonnement sur les développements. D'abord nous donnons une interprétation à la notion de développement de manière à pouvoir formellement discuter de la correction des développements. Nous nous appuyons ensuite sur cette interprétation pour définir une logique de raisonnement sur les développements. Et enfin nous examinons la correction ainsi que la complétude de cette logique de raisonnement.

6.4.1 Interprétation des développements

À chaque développement D nous associons une abstraction $\varphi_a(D)$ et une concrétisation $\varphi_c(D)$. L'abstraction $\varphi_a(D)$ est un programme abstrait qui permet de raisonner sur D en s'affranchissant de certains détails. La concrétisation $\varphi_c(D)$ est le programme concret qui résulte du processus de développement, et que l'on pourra éventuellement exécuter s'il est suffisamment concret. Ceci revient à associer à D le bloc suivant : $\varphi_a(D)$ { $\varphi_c(D)$ }. Prenons par exemple le développement suivant :

Exemple 15 (Permutation de variables)**.**

$$
\begin{array}{l}
\mathit{Swap} \stackrel{\text{def}}{=} (x, y) := (y, x) \ \{ \\
\quad \langle\ x' = y \ \wedge \ y' = x\ \rangle \ \ \{ \\
\qquad x := x - y;\ y := x + y;\ x := y - x \\
\quad \} \\
\}
\end{array}
$$

L'abstraction de ce développement est la spécification la plus abstraite qui le décrit, soit

$$\varphi_a(\mathsf{Swap}) \stackrel{\text{def}}{=} (x, y) := (y, x)$$

La concrétisation de Swap est le programme le plus concret qui décrit le développement,

soit

$$\varphi_c(\mathsf{Swap}) \stackrel{\text{def}}{=} x := x - y;\ y := x + y;\ x := y - x$$

Les fonctions φ_a et φ_c sont définies comme suit.

Définition 21 (Abstractions et concrétisations de développements)**.** *Soit D un développement. L'abstraction φ_a et la concrétisation φ_c de D représentent respectivement la première et la seconde projection de la fonction φ suivante :*

$$\begin{array}{rccl} \varphi: & \mathsf{Dev}^{T,T',b} & \longrightarrow & \mathsf{Stmt}^{T,T'} \times \mathsf{Stmt}^{T,T'} \\ & D & \longrightarrow & \varphi(D) = (\varphi_a(D),\ \varphi_c(D)) \end{array}$$

Ci-dessous, nous définissons φ par induction sur la syntaxe des développements. Pour marquer la relation entre cette définition et la notion de bloc spécifié, nous utilisons la notation des blocs spécifiés ($\varphi_a(D)$ { $\varphi_c(D)$ }) pour désigner la paire ($\varphi_a(D)$, $\varphi_c(D)$).

$$\begin{array}{ll} \varphi(\textbf{effect}\ f) & \stackrel{\text{def}}{=} (\textbf{effect}\ f)\ \{\ \ (\textbf{effect}\ f)\ \ \} \\ \varphi(\langle R\rangle) & \stackrel{\text{def}}{=} \langle R\rangle\ \{\ \ \langle R\rangle\ \ \} \\ \varphi(D_1;D_2) & \stackrel{\text{def}}{=} \varphi_a(D_1)\ ;\ \varphi_a(D_2)\ \{\ \ \varphi_c(D_1)\ ;\ \varphi_c(D_2)\ \ \} \\ \varphi(\textbf{if}\ C\ \textbf{then}\ D_1\ \textbf{else}\ D_2\ \textbf{end}) & \stackrel{\text{def}}{=} \textbf{if}\ C\ \textbf{then}\ \varphi_a(D_1)\ \textbf{else}\ \varphi_a(D_2)\ \textbf{end}\ \{ \\ & \qquad \textbf{if}\ C\ \textbf{then}\ \varphi_c(D_1)\ \textbf{else}\ \varphi_c(D_2)\ \textbf{end} \\ & \quad \} \\ \varphi(D_1\ \{\ D_2\ \}) & \stackrel{\text{def}}{=} \varphi_a(D_1)\ \{\ \ \varphi_c(D_2)\ \ \} \\ \varphi(\textbf{while}\ C\ \textbf{do}\ D\ \textbf{end}) & \stackrel{\text{def}}{=} \textbf{while}\ C\ \textbf{do}\ \varphi_a(D)\ \textbf{end}\ \{ \\ & \qquad \textbf{while}\ C\ \textbf{do}\ \varphi_c(D)\ \textbf{end} \\ & \quad \} \end{array}$$

Dans le cas des constructions de base où D est de la forme (**effect** f) ou $\langle R\rangle$, la seule abstraction dont on dispose et qui nous permet de raisonner sur D est D lui même. De plus D est aussi la seule concrétisation de D dont on dispose, donc $\varphi_a(D) = \varphi_c(D) \cong D$.

Lorsque D est un bloc spécifié, on considère que le programme le plus concret que l'on peut extraire de D correspond à la concrétisation de son corps D_2 étant donné que D_2 a pour but de décrire une implémentation de D_1 plus concrète. L'abstraction de D correspond naturellement à sa spécification D_1. En effet on montre aisément que $D_1 \cong \varphi_a(D_1) = \varphi_c(D_1)$ puisque D_1 est de type $\mathsf{Dev}^{T,T',\mathsf{true}}$.

Si D est une séquence, une alternative ou une boucle, l'abstraction de D ainsi que sa concrétisation s'obtiennent récursivement à partir des abstractions et des concrétisations

des développements qui le constituent. Dans le cas des abstractions, les appels récursifs ont pour effet d'ignorer les contenus des blocs. Et dans le cas des concrétisations, les appels récursifs ont pour effet d'ignorer les raffinements intermédiaires.

6.4.2 Raisonnement sur les développements

Muni d'une interprétation formelle des développements, nous allons maintenant formaliser une logique qui nous permet de raisonner sur les développements dans le but d'établir leur correction. Nous commencerons donc par définir ce qu'est un développement *correct*. Considérant qu'un développement D abouti mène à un programme exécutable $\varphi_c(D)$ qui satisfait la spécification de départ $\varphi_a(D)$, nous définissons les développements corrects comme suit.

Définition 22 (Développement correct)**.** *Un développement D est dit correct si et seulement si $\varphi_c(D) \sqsubseteq \varphi_a(D)$.*

De par cette définition, un développement peut être correct alors que ce dernier contient des étapes de raffinements incorrects. Considérons par exemple le développement suivant :

Exemple 16 (Développement erroné)**.**

$$
\begin{array}{rcl}
D & \stackrel{\text{def}}{=} & \textsf{skip}\ \{ \\
 & & \quad \textsf{abort}\ \{ \\
 & & \quad\quad \textsf{skip} \\
 & & \quad \}\}
\end{array}
$$

Dans D, la première étape de raffinement a consisté à raffiner **skip** avec **abort**. Ce raffinement est incorrect puisque **abort** $\sqsubseteq$ **skip** est faux. Par contre, le second raffinement est correct puisque **skip** $\sqsubseteq$ **abort** est vrai. Bien que D contient une étape de raffinement erronée, D est quand même correct puisque $\varphi_c(D) =$ **skip** $\sqsubseteq$ **skip** $= \varphi_a(D)$. Ceci est indésirable dans la mesure où un tel développement rendrait alors compte d'un processus de réflexion erroné. Nous distinguons donc les développements corrects de ceux dits *corrects par construction*, c'est-à-dire ceux dont la correction découle de la correction de toutes les étapes de raffinement.

Définition 23 (Développement correct par construction)**.** *Un développement D est dit correct par construction (ou CbC) si et seulement si (CbC D) est dérivable à partir des règles d'inférence de la figure 6.3 ci-dessous.*

$$\frac{}{\mathsf{CbC}\ (\textbf{effect}\ f)}\ (\mathsf{cbc\text{-}effect}) \qquad \frac{}{\mathsf{CbC}\ \langle R\rangle}\ (\mathsf{cbc\text{-}spec}) \qquad \frac{\mathsf{CbC}\ D_1\ \wedge\ \mathsf{CbC}\ D_2}{\mathsf{CbC}\ D_1;D_2}\ (\mathsf{cbc\text{-}seq})$$

$$\frac{\mathsf{CbC}\ D_1\ \wedge\ \mathsf{CbC}\ D_2}{\mathsf{CbC}\ (\textbf{if}\ C\ \textbf{then}\ D_1\ \textbf{else}\ D_2\ \textbf{end})}\ (\mathsf{cbc\text{-}if}) \qquad \frac{\mathsf{CbC}\ D_2\ \wedge\ \varphi_a(D_2) \sqsubseteq \varphi_a(D_1)}{\mathsf{CbC}\ (D_1\ \{\ D_2\ \})}\ (\mathsf{cbc\text{-}bloc})$$

$$\frac{\begin{array}{l}\mathsf{CbC}\ D\\ \wedge\ K;K\ \sqsubseteq\ K, \quad \text{avec } K \stackrel{\text{def}}{=} (\textbf{if}\ C\ \textbf{then}\ \varphi_a(D)\ \textbf{end})\\ \wedge\ \mathsf{wfd}\ (\lambda\ e\ e' \Rightarrow C\ e' \wedge (\llbracket\varphi_a(D)\rrbracket\ e'\ e) \wedge C\ s)\end{array}}{\mathsf{CbC}\ (\textbf{while}\ C\ \textbf{do}\ D\ \textbf{end})}\ (\mathsf{cbc\text{-}while})$$

Figure 6.3: Règles d'inférence des développements corrects par construction

Affectation et spécification. Les règles (cbc-effect) et (cbc-spec) sont en quelque sorte les axiomes du système de preuve. En effet, les instructions de base (**effect** f) et $\langle R\rangle$ sont interprétées comme représentant respectivement les étapes de raffinement (**effect** f) $\sqsubseteq$ (**effect** f) et $\langle R\rangle \sqsubseteq \langle R\rangle$. Ils sont donc corrects par construction étant donné qu'ils représentent exactement une étape triviale de raffinement.

Séquence et alternative. La règle (cbc-seq) permet la composition séquentielle de deux développements corrects par construction. Remarquons que les raffinements contenus dans $D_1;D_2$ sont exactement ceux contenus dans D_1 ou D_2. On peut donc considérer que $D_1;D_2$ ne contient pas de raffinements incorrects dès lors que ni D_1 ni D_2 n'en contiennent. Ceci est justement le cas lorsque D_1 comme D_2 sont corrects par construction, d'où les hypothèses $\mathsf{CbC}\ D_1$ et $\mathsf{CbC}\ D_2$. La règle (cbc-if) est analogue à la règle (cbc-seq) et se justifie de la même manière.

Bloc spécifié. La règle (cbc-bloc) permet sous deux conditions de dériver la correction par construction d'un bloc spécifié $D = D_1\ \{\ D_2\ \}$. La première condition concerne le corps du bloc, qui doit être correct par construction. Dès lors, on peut conclure que tous les raffinements contenus dans le bloc sont corrects, à l'exception peut-être du raffinement à l'origine de la création du bloc. La seconde condition qui requiert que l'abstraction $\varphi_a(D_2)$ du corps du bloc raffine la spécification du bloc $\varphi_a(D_1)$ permet de garantir la correction du raffinement à l'origine du bloc. Nous pouvons nous contenter de raisonner sur D_2 en utilisant l'abstraction de $\varphi_a(D_2)$ grâce à l'hypothèse que D_2 est correcte par construction. Ici, $\varphi_a(D_1)$ est équivalent à D_1 puisque D_1 est du type $\mathsf{Dev}^{T,T',\mathsf{true}}$ des développements qui ne contiennent pas de blocs. L'idée est que le développement D_2 s'interprète comme le bloc $\varphi_a(D_2)\ \{\ ...\ \}$. Par conséquent on peut imaginer que le bloc D a d'abord été créé en

écrivant D_1 { $\varphi_a(D_2)$ }, avant de devenir $D = D_1$ { $\varphi_a(D_2)$ { ... } }. Donc, à la création du bloc, le raffinement à prouver correspond bien à la seconde hypothèse $\varphi_a(D_2) \sqsubseteq \varphi_a(D_1)$.

Notons que quelque soit l'état d'un développement, la transformation d'une instruction en bloc spécifié n'a aucun impact sur les énoncés des raffinements précédents ni les preuves déjà effectuées. En effet, pour tout D_x de type $\mathsf{Dev}^{T,T',\mathsf{true}}$, les énoncés des raffinements déjà effectués dépendent de $\varphi_a(D_x)$. Par conséquent, lorsque D_x devient $D = D_x$ { ... } les énoncés précédents qui dépendront désormais de $\varphi_a(D)$ ne changent pas puisque, par définition de φ, on a $\varphi_a(D_x) = \varphi_a(D)$.

Enfin, précisons que le système de dérivation n'impose aucune règle quant à l'ordre dans lequel il faut prouver les différents raffinements. Il exige seulement qu'ils soient tous établis à un moment donné. Le programmeur a donc la possibilité de retarder les preuves de raffinement.

Boucle. Pour qu'une boucle ne contienne pas de raffinements incorrects, il suffit que son corps D n'en contienne pas. Donc a priori la règle (cbc-while) pourrait avoir comme seul antécédent ($\mathsf{Cbc}\ D$). Cependant, comme nous l'avons vu précédemment, le raisonnement sur les boucles n'est pas simple à faire directement avec la sémantique. Pour faciliter les raisonnements sur les boucles, la règle (cbc-while) requiert aussi deux autres conditions qui sont les conditions suffisantes pour que le lemme d'abréviation des boucles (lemme 16) soit applicable. Comme dans le cas des blocs spécifiés, nous pouvons nous limiter à vérifier que les conditions du lemme d'abréviation sont effectivement valides pour l'abstraction $\varphi_a(D)$ grâce à l'hypothèse ($\mathsf{CbC}\ D$). Il s'ensuit d'une part une facilitation de la lecture des développements. Par exemple, à la lecture d'un développement correct de boucle, on sait que ce dernier peut s'abréger en alternative selon l'équivalence suivante :

$$\textbf{while}\ C\ \textbf{do}\ \langle R \rangle\ \{\ ...\ \}\ \textbf{end} \quad \equiv \quad \textbf{if}\ C\ \textbf{then}\ \langle\ \lambda\ e\ e' \Rightarrow R\ e\ e' \wedge \neg C\ e'\ \rangle\ \textbf{end}$$

D'autre part, cette même équivalence permet de faciliter les preuves de raffinements par application du théorème 6 qui permet de faire passer les énoncés à démontrer de la logique du second ordre à la logique du premier ordre. L'applicabilité du lemme d'abréviation a aussi pour conséquence le théorème suivant qui facilite la gestion de l'opérateur (;) dans le cas par exemple où l'abstraction $\varphi_a(D)$ du corps d'une boucle est de la forme $S_1;S_2;...;S_n$.

Théorème 7.

$$\begin{array}{l} \forall\ C\ D\ R\ \cdot\ \mathit{CbC}\ (\textbf{while}\ C\ \textbf{do}\ D\ \textbf{end}) \\ \rightarrow \begin{array}{l} \mathit{wpr}\ (\textbf{while}\ C\ \textbf{do}\ \varphi_a(D)\ \textbf{end})\ R \\ \equiv\ \mathit{wpr}\ (\textbf{if}\ C\ \textbf{then}\ \varphi_a(D)\ \textbf{end})\ (\lambda\ e\ e' \Rightarrow \neg(C\ e') \rightarrow R\ e\ e') \end{array} \end{array}$$

Schéma de preuve. Après application du lemme d'abréviation, il suffit de montrer que :

$$\begin{aligned}&\mathsf{wpr}\ (\textbf{if}\ C\ \textbf{then}\ \langle\ \lambda\ e\ e' \Rightarrow [\![\varphi_a(D)]\!]\ e\ e'\ \wedge \neg(C\ e')\rangle\ \textbf{end})\ R)\\ \equiv\ &\mathsf{wpr}\ (\textbf{if}\ C\ \textbf{then}\ \varphi_a(D)\ \textbf{end})\ (\lambda\ e\ e' \Rightarrow \neg(C\ e') \rightarrow R\ e\ e')\end{aligned}$$

Ensuite la preuve consiste en une suite de raisonnements assez simples. □

6.4.3 Correction

Nous allons maintenant montrer que tout développement D correct par construction est en effet correct dans le sens où le programme $\varphi_c(D)$ que l'on en tire implémente la spécification de haut niveau $\varphi_a(D)$. Formellement, il s'agit d'établir le théorème 8 ci-dessous qui est principalement une conséquence de la monotonie des structures de contrôle vis-à-vis de la relation de raffinement. À l'issue d'un développement D, si nous savons dériver ($\mathsf{CbC}\ D$), alors nous pouvons appliquer ce théorème pour obtenir un λ-terme dont le type est $\varphi_c(D) \sqsubseteq \varphi_a(D)$. Ce λ-terme est un certificat qui garantit que le programme associé implémente bien la spécification associée. Le fait que le certificat a bien le type attendu peut être établi mécaniquement et indépendamment du processus de développement mis en œuvre.

Théorème 8 (Correction)**.** $\forall\ D \cdot \mathsf{CbC}\ D \rightarrow \varphi_c(D) \sqsubseteq \varphi_a(D)$

Preuve. On procède par induction sur la dérivation $\mathsf{CbC}\ D$.

- Cas (cbc-effect) :

  $\varphi_c(\textbf{effect}\ f)\ \sqsubseteq\ \varphi_a(\textbf{effect}\ f)$
  $\leftarrow$ $\textbf{effect}\ f\ \sqsubseteq\ \textbf{effect}\ f$ (par définition de φ)
  $\leftarrow$ $\top$ (par réflexivité de $\sqsubseteq$ (lemme 8))
  $\leftarrow$ $\mathsf{CbC}\ (\textbf{effect}\ f)$ (par élimination de (cbc-effect))

- Cas (cbc-spec) :

  $\varphi_c(\langle\ R\ \rangle)\ \sqsubseteq\ \varphi_a(\langle\ R\ \rangle)$
  $\leftarrow$ $\langle\ R\ \rangle\ \sqsubseteq\ \langle\ R\ \rangle$ (par définition de φ)
  $\leftarrow$ $\top$ (par réflexivité de $\sqsubseteq$ (lemme 8))
  $\leftarrow$ $\mathsf{CbC}\ \langle\ R\ \rangle$ (par élimination de (cbc-spec))

- Cas (cbc-seq) :

$\varphi_c(D_1;D_2) \sqsubseteq \varphi_a(D_1;D_2)$
$\leftarrow \varphi_c(D_1);\varphi_c(D_2) \sqsubseteq \varphi_a(D_1);\varphi_a(D_2)$ (par définition de φ)
$\leftarrow \varphi_c(D_1) \sqsubseteq \varphi_a(D_1) \wedge \varphi_c(D_2) \sqsubseteq \varphi_a(D_2)$ (par monotonie de ; (lemme 9))
$\leftarrow \mathsf{CbC}\ D_1 \wedge \mathsf{CbC}\ D_2$ (par hypothèse d'induction)
$\leftarrow \mathsf{CbC}\ (D_1;D_2)$ (par élimination de (cbc-seq))

- Cas (cbc-if) :

$\varphi_c(\textbf{if}\ C\ \textbf{then}\ D_1\ \textbf{else}\ D_2\ \textbf{end}) \sqsubseteq$
$\varphi_a(\textbf{if}\ C\ \textbf{then}\ D_1\ \textbf{else}\ D_2\ \textbf{end})$

$\leftarrow$ $\textbf{if}\ C\ \textbf{then}\ \varphi_c(D_1)\ \textbf{else}\ \varphi_c(D_2)\ \textbf{end} \sqsubseteq$ $\textbf{if}\ C\ \textbf{then}\ \varphi_a(D_1)\ \textbf{else}\ \varphi_a(D_2)\ \textbf{end}$ (par définition de φ)

$\leftarrow \varphi_c(D_1) \sqsubseteq \varphi_a(D_1) \wedge \varphi_c(D_2) \sqsubseteq \varphi_a(D_2)$ (par monotonie de **if** (lemme 9))
$\leftarrow \mathsf{CbC}\ D_1 \wedge \mathsf{CbC}\ D_2$ (par hypothèse d'induction)
$\leftarrow \mathsf{CbC}\ (\textbf{if}\ C\ \textbf{then}\ D_1\ \textbf{else}\ D_2\ \textbf{end})$ (par élimination de (cbc-if))

- Cas (cbc-bloc) :

$\varphi_c(D_1\ \{\ D_2\ \}) \sqsubseteq \varphi_a(D_1\ \{\ D_2\ \})$
$\leftarrow \varphi_c(D_2) \sqsubseteq \varphi_a(D_1)$ (par définition de φ)
$\leftarrow \varphi_c(D_2) \sqsubseteq \varphi_a(D_2) \wedge \varphi_a(D_2) \sqsubseteq \varphi_a(D_1)$ (par transitivité de $\sqsubseteq$ (lemme 8))
$\leftarrow \mathsf{CbC}\ D_2 \wedge \varphi_a(D_2) \sqsubseteq \varphi_a(D_1)$ (par hypothèse d'induction)
$\leftarrow \mathsf{CbC}\ (D_1\ \{\ D_2\ \})$ (par élimination de (cbc-bloc))

- Cas (cbc-while) :

$\varphi_c(\textbf{while}\ C\ \textbf{do}\ D\ \textbf{end}) \sqsubseteq$
$\varphi_a(\textbf{while}\ C\ \textbf{do}\ D\ \textbf{end})$

$\leftarrow$ $\textbf{while}\ C\ \textbf{do}\ \varphi_c(D)\ \textbf{end} \sqsubseteq$ $\textbf{while}\ C\ \textbf{do}\ \varphi_a(D)\ \textbf{end}$ (par définition de φ)

$\leftarrow \varphi_c(D) \sqsubseteq \varphi_a(D)$ (par monotonie de **while** (lemme 9))
$\leftarrow \mathsf{CbC}\ D$ (par hypothèse d'induction)
$\leftarrow \mathsf{CbC}\ (\textbf{while}\ C\ \textbf{do}\ D\ \textbf{end})$ (par élimination de (cbc-while))

□

Le lecteur aura noté que le système de preuve correspondant à notre définition de la correction par construction est dirigé par la syntaxe. Comme Parent le fait remarquer dans [63],

les programmes tirés d'une preuve de correction laissent entrevoir le squelette de cette preuve. D'une certaine manière, ce phénomène transparaît dans notre formalisation. En effet, on peut considérer que les constructions de notre langage de développement permettent d'appliquer les règles d'inférence du système de preuve. Ce faisant, l'arbre de la future preuve globale de correction se construit au fur et à mesure que le développement du programme progresse. Comme on peut le voir dans la preuve ci-dessus, la correction des règles (cbc-seq), (cbc-if) et (cbc-while) découle de la monotonie des constructions correspondantes vis-à-vis de la relation de raffinement. De même, la correction de (cbc-bloc) découle de la transitivité de la relation de raffinement. Par conséquent, la programmation d'une séquence, d'une sélection, ou d'une boucle, de même que la création d'un bloc spécifié, consistent aussi à profiter de l'occasion pour insérer, dans l'arbre de preuve, l'application de la propriété de monotonie de ces structures ou de la propriété de transitivité de la relation de raffinement.

6.4.4 Complétude

Comme nous l'avons vu précédemment, il est possible d'avoir $\varphi_c(D) \sqsubseteq \varphi_a(D)$ alors que CbC D n'est pas dérivable. Donc, comme on pourrait s'y attendre, les règles d'inférence ne sont pas complètes au sens absolu du terme. Cependant, nous pouvons montrer le théorème 9 ci-dessous qui reflète le fait que les règles sont complètes au sens plus relatif où, pour toute concrétisation S_2 et abstraction S_1 telles que $S_2 \sqsubseteq S_1$, il existe un développement correct par construction D dont la concrétisation et l'abstraction associées sont respectivement S_2 et S_1.

Théorème 9 (Complétude)**.** $\forall\ S_2\ S_1 \cdot\ S_2 \sqsubseteq S_1 \rightarrow \exists\ D \cdot \mathit{CbC}\ D\ \wedge\ \varphi(D) = S_1\ \{\ S_2\ \}$

Preuve. On procède par induction sur S_2.

- Cas $S_2 = (\textbf{effect}\ f)$:

$$\begin{array}{lll} & \mathsf{CbC}\ (S_1\ \{\ (\textbf{effect}\ f)\ \}) & \\ \wedge & \varphi(S_1\ \{\ (\textbf{effect}\ f)\ \}) = S_1\ \{\ (\textbf{effect}\ f)\ \} & \\ & & \\ \leftarrow & \mathsf{CbC}\ (S_1\ \{\ (\textbf{effect}\ f)\ \}) & \\ \wedge & \varphi_a(S_1)\ \{\ \varphi_c(\textbf{effect}\ f)\ \} = S_1\ \{\ (\textbf{effect}\ f)\ \} & \text{(par définition de } \varphi) \\ & & \\ \leftarrow & \mathsf{CbC}\ (S_1\ \{\ (\textbf{effect}\ f)\ \}) & \\ \wedge & S_1\ \{\ (\textbf{effect}\ f)\ \} = S_1\ \{\ (\textbf{effect}\ f)\ \} & \text{(par définition de } \varphi) \end{array}$$

$\leftarrow$ $\mathsf{CbC}\ (S_1\ \{\ (\textbf{effect}\ f)\ \})$

$\leftarrow$ $\mathsf{CbC}\ (\textbf{effect}\ f)\ \wedge\ \textbf{effect}\ f \sqsubseteq S_1$ (par (cbc-bloc))

$\leftarrow$ $\textbf{effect}\ f \sqsubseteq S_1$ (par (cbc-effect))

- Cas $S_2 = \langle\ R\ \rangle$:

  $\mathsf{CbC}\ (S_1\ \{\ \langle\ R\ \rangle\ \})\ \wedge\ \varphi(S_1\ \{\ \langle\ R\ \rangle\ \}) = S_1\ \{\ \langle\ R\ \rangle\ \}$

  $\leftarrow$ $\mathsf{CbC}\ (S_1\ \{\ \langle\ R\ \rangle\ \})\ \wedge\ \varphi_a(S_1)\ \{\ \varphi_c(\langle\ R\ \rangle)\ \} = S_1\ \{\ \langle\ R\ \rangle\ \}$ (par définition de φ)

  $\leftarrow$ $\mathsf{CbC}\ (S_1\ \{\ \langle\ R\ \rangle\ \})\ \wedge\ S_1\ \{\ \langle\ R\ \rangle\ \} = S_1\ \{\ \langle\ R\ \rangle\ \}$ (par déf. de φ)

  $\leftarrow$ $\mathsf{CbC}\ (S_1\ \{\ \langle\ R\ \rangle\ \})$

  $\leftarrow$ $\mathsf{CbC}\ \langle\ R\ \rangle\ \wedge\ \langle\ R\ \rangle \sqsubseteq S_1$ (par (cbc-bloc))

  $\leftarrow$ $\langle\ R\ \rangle \sqsubseteq S_1$ (par (cbc-spec))

- Cas $S_2 = S_{21}\ ;\ S_{22}$:

  $\leftarrow$ $\mathsf{CbC}\ (S_1\ \{\ D_{21}\ ;\ D_{22}\ \}) \wedge$
  $\varphi(S_1\ \{\ D_{21}\ ;\ D_{22}\ \}) = S_1\ \{\ S_{21}\ ;\ S_{22}\ \}$

  $\leftarrow$ $\mathsf{CbC}\ (D_{21}\ ;\ D_{22})\ \wedge\ \varphi_a(D_{21})\ ;\ \varphi_a(D_{22})\ \sqsubseteq\ S_1$
  $\wedge\ \varphi_c(D_{21}\ ;\ D_{22}) = S_{21}\ ;\ S_{22}$ (par (cbc-bloc) et déf. de φ)

  $\leftarrow$ $\mathsf{CbC}\ D_{21}\ \wedge\ \varphi(D_{21}) = S_{21}\ \{\ S_{21}\ \}$
  $\wedge\ \mathsf{CbC}\ D_{22}\ \wedge\ \varphi(D_{22}) = S_{22}\ \{\ S_{22}\ \}$ (par (cbc-seq) et déf. de φ)
  $\wedge\ S_{21}\ ;\ S_{22}\ \sqsubseteq\ S_1$

  $\leftarrow$ $S_{21}\ ;\ S_{22}\ \sqsubseteq\ S_1$ (par hypothèse d'induction)

- Cas $S_2 = \textbf{if}\ C\ \textbf{then}\ S_{21}\ \textbf{else}\ S_{22}\ \textbf{end}$:

  $\leftarrow$ $\mathsf{CbC}\ (S_1\ \{\ \textbf{if}\ C\ \textbf{then}\ D_{21}\ \textbf{else}\ D_{22}\ \textbf{end}\ \}) \wedge$
  $\varphi(S_1\ \{\ \textbf{if}\ C\ \textbf{then}\ D_{21}\ \textbf{else}\ D_{22}\ \textbf{end}\ \}) = S_1\ \{\ S_2\ \}$

  $\leftarrow$ $\mathsf{CbC}\ (\textbf{if}\ C\ \textbf{then}\ D_{21}\ \textbf{else}\ D_{22}\ \textbf{end})$
  $\wedge\ \textbf{if}\ C\ \textbf{then}\ \varphi_a(D_{21})\ \textbf{else}\ \varphi_a(D_{22})\ \textbf{end}\ \sqsubseteq\ S_1$ (par (cbc-bloc) et déf. de φ)
  $\wedge\ \varphi_c(\textbf{if}\ C\ \textbf{then}\ D_{21}\ \textbf{else}\ D_{22}\ \textbf{end}) = S_2$

  $\leftarrow$ $\mathsf{CbC}\ D_{21}\ \wedge\ \varphi(D_{21}) = S_{21}\ \{\ S_{21}\ \}$
  $\wedge\ \mathsf{CbC}\ D_{22}\ \wedge\ \varphi(D_{22}) = S_{22}\ \{\ S_{22}\ \}$ (par (cbc-if) et déf. de φ)
  $\wedge\ \textbf{if}\ C\ \textbf{then}\ S_{21}\ \textbf{else}\ S_{22}\ \textbf{end}\ \sqsubseteq\ S_1$

  $\leftarrow$ $\textbf{if}\ C\ \textbf{then}\ S_{21}\ \textbf{else}\ S_{22}\ \textbf{end}\ \sqsubseteq\ S_1$ (par hypothèse d'induction)

- Cas $S_2 =$ **while** C **do** S **end** :

$$
\begin{array}{ll}
 & \mathsf{CbC}\ (S_1\ \{\ \textbf{while}\ C\ \textbf{do}\ D\ \textbf{end}\ \}) \\
\leftarrow & \wedge\ \varphi_a(S_1\ \{\ \textbf{while}\ C\ \textbf{do}\ D\ \textbf{end}\ \}) = S_1 \\
 & \wedge\ \varphi_c(S_1\ \{\ \textbf{while}\ C\ \textbf{do}\ D\ \textbf{end}\ \}) = \textbf{while}\ C\ \textbf{do}\ S\ \textbf{end}
\end{array}
$$

$$
\begin{array}{lll}
 & \mathsf{CbC}\ (\textbf{while}\ C\ \textbf{do}\ D\ \textbf{end}) & \\
\leftarrow & \wedge\ \varphi_a(\textbf{while}\ C\ \textbf{do}\ D\ \textbf{end}) \ \sqsubseteq\ S_1 & (\ \text{par (cbc-bloc)} \\
 & \wedge\ \varphi(D) = K\ \{\ S\ \} & \text{et par déf. de } \varphi\ )
\end{array}
$$

$$
\begin{array}{lll}
 & \mathsf{CbC}\ D\ \wedge\ \varphi(D) = K\ \{\ S\ \} & \\
 & \wedge\ \mathsf{wfd}\ (\lambda\ e\ e' \Rightarrow C\ e' \wedge [\![\varphi_a(D)]\!]\ e'\ e \wedge C\ e) & \\
\leftarrow & \wedge\ \mathsf{let}\ L \stackrel{\text{def}}{=} \textbf{if}\ C\ \textbf{then}\ \varphi_a(D)\ \textbf{end}\ \mathsf{in}\ L\ ;\ L \sqsubseteq L & (\ \text{par (cbc-while)} \\
 & \wedge\ \textbf{while}\ C\ \textbf{do}\ \varphi_a(D)\ \textbf{end} \ \sqsubseteq\ S_1 & \text{et par déf. de } \varphi\ )
\end{array}
$$

$$
\begin{array}{lll}
 & \mathsf{CbC}\ D\ \wedge\ \varphi(D) = K\ \{\ S\ \} & \\
 & \wedge\ \mathsf{wfd}\ (\lambda\ e\ e' \Rightarrow C\ e' \wedge [\![K]\!]\ e'\ e \wedge C\ e) & \\
\leftarrow & \wedge\ \mathsf{let}\ L \stackrel{\text{def}}{=} \textbf{if}\ C\ \textbf{then}\ K\ \textbf{end}\ \mathsf{in}\ L\ ;\ L \sqsubseteq L & (\text{par définition de } \varphi) \\
 & \wedge\ \textbf{while}\ C\ \textbf{do}\ K\ \textbf{end} \ \sqsubseteq\ S_1 &
\end{array}
$$

$$
\begin{array}{lll}
 & S\ \sqsubseteq\ K & \\
 & \wedge\ \mathsf{wfd}\ (\lambda\ e\ e' \Rightarrow C\ e' \wedge [\![K]\!]\ e'\ e \wedge C\ e) & \\
\leftarrow & \wedge\ \mathsf{let}\ L \stackrel{\text{def}}{=} \textbf{if}\ C\ \textbf{then}\ K\ \textbf{end}\ \mathsf{in}\ L\ ;\ L \sqsubseteq L & (\ \text{hyp. d'induction} \\
 & \wedge\ \textbf{if}\ C\ \textbf{then}\ \langle\ \lambda\ e\ e' \Rightarrow [\![K]\!]\ e\ e'\ \wedge\ \neg(C\ e')\ \rangle\ \textbf{end}\ \sqsubseteq\ S_1 & \text{et par le lemme 16}\ )
\end{array}
$$

$$
\leftarrow\ \textbf{while}\ C\ \textbf{do}\ S\ \textbf{end} \ \sqsubseteq\ S_1 \qquad (\text{par le théorème 6})
$$

□

Ce théorème est essentiellement une conséquence de la complétude de notre méthode de preuve de raffinement pour les boucles (théorème 6). On en déduit que les règles de correction par construction ne limitent pas le type des programmes concrets qui peuvent être obtenus par application de ces règles.

6.5 Conclusion

Dans ce chapitre nous avons formalisé un langage de développement permettant de décrire un développement comme un enchaînement hiérarchique de raffinements. Cette formali-

sation permet d'une part d'obtenir des développements plus intelligibles, et d'autre part de mieux maîtriser la complexité grâce à la possibilité de procéder de manière modulaire. Nous avons également formalisé une notion de correction par construction ainsi qu'une logique associée permettant d'établir qu'un développement donné est correct. Nous nous sommes assuré que cette logique de raisonnement est correcte et relativement complète, donc elle ne réduit pas l'expressivité du langage de programmation.

L'approche que nous avons formalisée est inspirée de travaux de Kourie et Watson [43] sur les méthodologies pour le développement correct par construction, et des travaux de Hehner [32] sur les blocs spécifiés. Dans ces travaux le langage étudié, qui supporte les procédures ou encore la programmation non structurée, est plus évolué que celui que nous avons considéré. En revanche, comme notre formalisation a été effectuée dans le cadre de la théorie des types, elle permet la production d'un certificat. Par conséquent, la vérification que le programme obtenu implémente effectivement la spécification initiale peut se faire mécaniquement par typage sans avoir à se préoccuper de l'environnement de développement utilisé, ni du processus de développement qui a été employé, ni des règles de raisonnements qui ont été appliquées.

À la différence des approches préconisées par Butler et al. [14] et Alpuim et al. [4], notre approche produit un objet plus intelligible car les différentes étapes de raffinements sont capturées hiérarchiquement dans le texte au lieu d'être embarquées dans la preuve de correction. De plus, elle est plus uniforme puisque toutes les spécifications sont simplement des relations. En particulier, les corps de boucles sont spécifiés de la même manière que tous les autres fragments de programmes.

Chapitre 7

Mécanisation en Coq

Contrairement à la démarche classique dans laquelle l'étape de la preuve intervient après l'étape du développement, la démarche par raffinements est par essence une démarche interactive dans laquelle plusieurs étapes de développement et de preuve s'entrelacent. Dans un tel contexte, l'utilisation d'un assistant de preuve comme outil d'aide au développement certifié semble tout à fait indiqué. En effet, en supposant que les programmes impératifs sont représentés, dans le langage de l'assistant de preuve, par les termes d'un type donné, il existe un parallèle évident entre la construction interactive et par étapes de termes de preuve, et l'élaboration par raffinements d'un programme impératif. De plus la théorie des types dépendants à la base de certains assistants de preuve permet de bénéficier immédiatement de la vérification du typage de nos constructions avec une possibilité d'inférence automatique pour certains types, ainsi que du polymorphisme paramétrique. Ce chapitre présente la mécanisation [1] de la théorie de la programmation par raffinements qui a été décrite dans les précédents chapitres. L'objectif de cette mécanisation est de permettre la certification de programmes impératifs grâce à la mise en œuvre de l'approche par raffinements dans l'assistant de preuve Coq. Après une brève présentation de Coq, nous exposerons la mécanisation en deux parties. La première partie concerne l'interface de la mécanisation qui est la partie de la mécanisation en laquelle l'utilisateur doit avoir confiance. Et la seconde partie concerne les mécanismes qui permettent à l'utilisateur de construire interactivement des développements corrects par construction. Enfin, avant de conclure, nous parlerons des travaux connexes liés à l'application de la démarche de raffinement dans Coq.

[1] https://github.com/bsall/thesis-dev

7.1 L'assistant de preuve Coq

Le système *Coq* permet au programmeur de spécifier, programmer et prouver dans un seul et même langage fonctionnel pur. Dans ce langage une proposition correspond à un type, et une preuve est un lambda terme. Vérifier la correction d'une preuve revient à établir que le terme de preuve a pour type la proposition associée. La vérification des preuves est effectuée par un composant du système dont la taille est relativement réduite par rapport à la taille du système. Il s'agit du *noyau* qui est le composant sur lequel repose la confiance qu'on peut avoir dans les résultats établis avec le système. Pour une introduction plus exhaustive de l'assistant de preuve Coq, nous renvoyons le lecteur à [65], et pour une présentation détaillée le lecteur pourra consulter [12, 15]. Ici, nous nous limitons à une brève présentation des types inductifs, des fonctions, et des outils permettant de construire des termes graduellement.

Types inductifs. La théorie à la base de Coq est le calcul des construction inductives (*CIC*) [19, 64] dans laquelle il existe une notion primitive de type inductif. Par exemple, l'exemple classique du type des listes polymorphes se définit comme suit :

`Inductive` *list* (A : `Type`) : `Type` :=
| *nil* : *list* A
| *cons* : $A \to$ *list* $A \to$ *list* A.

Le mot clé Inductive permet de déclarer les types inductifs. Ici A représente le type des éléments contenus dans la liste, et les deux constructeurs classiques permettent de construire respectivement un liste vide ou une nouvelle liste en ajoutant un élément en tête d'une liste existante. Pour prévenir la construction de types inductifs vides, les définitions inductive doivent respecter un critère dit de *positivité*. Dans l'exemple ci-dessus, ce critère interdit l'ajout d'un constructeur dont le type serait (list $A \to A$) $\to$ list A car le type en cours de définition (list A) apparaît en position négative, c'est-à-dire à gauche du symbole ($\to$).

Définitions. Le mot clé Definition permet de nommer des objets ou de définir des fonctions. Dans le cas des définitions récursives on emploie le mot clé Fixpoint. Par exemple, l'addition de deux entiers naturels peut se définir comme ci-dessous :

```
Fixpoint add (n m : nat) : nat :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (add n' m)
  end.
```

Cette définition de la fonction add illustre l'usage du filtrage sur un type inductif. Ici, le type nat est un type inductif à deux constructeurs : O pour le zéro, et S pour le successeur.

Outils pour le raffinement. Le système Coq permet de construire des termes d'un certain type graduellement grâce à la tactique **Program** [63, 71]. L'exemple ci-dessous illustre l'utilisation de cette tactique pour construire une fonction qui retourne le contenu d'une liste l, à l'indice n, sachant que n est bien un indice valide de l.

```
Require Import Coq.Program.Program Lia.

Program Fixpoint nth { A : Type } (l : list A) (n : nat) (p : n < length l) : A :=
  match l with
  | nil ⇒ _
  | cons a l ⇒ match n with
                | 0 ⇒ a
                | S n ⇒ nth l n _
                end
  end.
```

La première ligne de l'extrait ci-dessus importe deux librairies qui permettent respectivement de rendre accessible la tactique Program ainsi que d'autres tactiques permettant de raisonner sur l'arithmétique linéaire. Dans la définition de la fonction nth, le mot clé Program précède Fixpoint, ce qui permet d'indiquer au ssystème notre intention de précéder à la définition de manière graduelle. Cette définition procède d'abord par filtrage sur la liste l. Dans le cas d'une liste vide, nous somme en présence d'un cas impossible puisque si l est vide, alors on ne peut passer p parce que $n < 0$ n'est pas prouvable. Afin de ne pas encombrer notre définition, nous nous contentons d'écrire '_' pour indiquer que nous justifierons plus tard l'absence de terme pour le cas $l =$ nil. Dans le cas où la liste n'est pas vide, et si n est positif, un appel récursif s'impose, mais cet appel nécessite un paramètre p qui est un terme de preuve qui'il est plus facile de construire dans un autre contexte. On procède donc comme précédemment en écrivant simplement '_'. Notre définition est

lisible et permet de communiquer clairement nos intentions, cependant nous ne pourrons pas l'utiliser tant que nous ne l'aurons pas complété. Pour ce faire, il suffit d'exécuter la commande Next Obligation et ensuite de fournir le script de preuve permettant de construire le terme manquant. Dans notre cas, nous devrons exécuter deux fois cette commande. Par exemple, le script ci-dessous, permet de compléter notre définition dans un contexte plus adapté de preuve interactive.

```
Next Obligation. contradict p. simpl. lia. Qed.
Next Obligation. simpl in *. lia. Qed.
```

La première ligne procède par contradiction pour montrer qu'il est impossible de passer un terme p de type $n < 0$, et la seconde ligne permet de montrer automatiquement et grâce à la tactique lia la validité de la proposition suivante : $S\ n < (\mathsf{length}\ l) \rightarrow n < (\mathsf{length}\ l)$.

7.2 Spécifications

La première étape d'un développement consiste à formaliser la spécification du problème à résoudre. Dans le système Coq on considérera que les spécifications sont les termes du type des spécifications appelé Specification. La définition de ce type dans le langage de Coq est donnée ci-dessous :

```
Definition Specification { T T' : Type } := T → T' → Prop.
```

Le type des spécifications est le type des fonctions qui calculent une proposition à partir d'un état initial de type T, et d'un état final de type T'. Cette définition est différente de notre formalisation précédente dans laquelle le type des spécifications est : $T \rightarrow T' \rightarrow \mathsf{Type}$. Dans le système Coq, le type Type des types quelconques n'est pas clos par conjonction et disjonction, alors que le type Prop des propositions l'est. Nous choisissons donc d'utiliser le type Prop pour exprimer les spécifications car cela facilite leur composition, et cela permet aussi de tirer parti de certaines tactiques de raisonnement prédéfinies pour les termes de type Prop, telles que par exemple les tactiques split, left, right, intuition, firstorder, etc.. Pour illustration, la spécification ci-dessous formalise le problème de la permutation de deux variables.

```
Definition Swap { T : Type } : @Specification (T × T) (T × T) :=
  fun (e e' : T×T) ⇒
  match (e,e') with
    ((x,y),(x',y')) ⇒ x' = y ∧ y' = x
  end.
```

Ici, l'état initial e, comme l'état final e', est constitué de deux variables de type T. La dé-construction par filtrage des états e et e', expose les variables x,y,x' et y' qui constituent ces états. Dès lors, on peut retourner la proposition $x' = y \wedge y' = x$ qui exprime le comportement voulu. Il est possible d'exprimer cette spécification de manière plus concise comme suit :

```
Definition Swap { T } : @Specification (T × T) _ :=
  fun '(x,y) '(x',y') ⇒ x' = y ∧ y' = x.
```

Dans la spécification alternative ci-dessus, on tire avantage de la capacité de Coq à inférer les types de T et du second paramètre de la définition `@Specification`. De plus la notation $'(x, y)$ (respectivement $'(x', y')$) permet d'économiser la déclaration de l'état initial e (respectivement l'état final e') et sa future dé-construction par filtrage. La concision des spécifications peut aussi être améliorée par l'utilisation de définitions auxiliaires. Étant donné que le système Coq permet de programmer et de spécifier avec le même langage, l'utilisation des définitions auxiliaires est complètement naturelle. Par exemple, la spécification ci-dessous du problème du calcul de la racine carrée entière utilise la définition auxiliaire `sqr` du carré d'un entier. Dans ce cas précis, le type des états initiaux et finaux peut être inféré, la spécification peut donc s'écrire de manière encore plus concise.

```
Definition sqr n := n × n.
Definition Sqrt : Specification :=
    (fun x '(x',r') ⇒ sqr r' ≤ x <  sqr (1 + r') ∧ x = x').
```

7.3 Programmes et raffinement

Étant donné qu'il s'agit d'élaborer un programme concret qui implémente une spécification abstraite pouvant aussi être considérée comme un programme, nous devons préciser les éléments vis-à-vis desquels la relation d'implémentation doit se comprendre. Ces éléments, qui sont au nombre de trois, forment l'interface externe de notre mécanisation.

7.3.1 Syntaxe et interprétations des programmes

Le premier de ces éléments est la syntaxe du langage impératif étudié. La syntaxe abstraite de ce langage qui a été spécifiée plus haut (figure 4.1 en page 32) se matérialise en Coq par le type inductif ci-dessous.

```
Inductive Stmt { T } : ∀ (T' : Type), Type :=
| Assignment { T' } : (T → T') → @Stmt T T'
| Seq : ∀ { U T' }, @Stmt T U → @Stmt U T' → @Stmt T T'
| If { T' } : (T → Prop) → @Stmt T T' → @Stmt T T' → @Stmt T T'
| While : (T → Prop) → @Stmt T T → @Stmt T T
| Spec { T' } : @Specification T T' → @Stmt T T'.
```

Le second élément d'interface est l'interprétation prédicative que nous associons à notre langage de programmation. Cette interprétation correspond à la sémantique prédicative spécifiée dans la figure 4.2 (page 34). Dans la mécanisation en Coq, l'interprétation des programmes est représentée par la fonction pred décrite ci-dessous.

```
Fixpoint pred { T V : Type } (C : @Stmt T V) { struct C }
: Specification :=
  match C with
  | Assignment f ⇒ (fun s s' ⇒ f s = s')
  | Seq c1 c2 ⇒ ⟦c1⟧ □ ⟦c2⟧
  | If p ct cf ⇒ (fun s s' ⇒ (p s ∧ ⟦ct⟧ s s') ∨ ( ¬p s ∧ ⟦cf⟧ s s'))
  | While p c ⇒ lfp (fun X s s' ⇒ p s ∧ (⟦c⟧ □ X) s s' ∨ ¬p s ∧ s = s' )
  | Spec spec ⇒ spec
  end
where "⟦ c ⟧" := (pred c).
```

L'interprétation de la séquence et de la boucle est basée sur l'opérateur de composition démoniaque (définition 13 en page 35) codifiée comme suit en Coq.

```
Definition dcomp {T U T' : Type}
(P : @Specification T U) (Q : @Specification U T') : @Specification T T' :=
fun s s' ⇒ (∃ sx, P s sx ∧ Q sx s') ∧ (∀ sx, P s sx → ∃ s', Q sx s').
Notation "P □ Q" := (dcomp P Q) (at level 90, right associativity, ...).
```

On remarquera que dans cette définition, le terme ($P\ s\ sx$) apparaît en position négative dans le membre de droite de la conjonction. Par conséquent, il n'est pas possible de codifier l'interprétation *pred* classiquement avec un type inductif en raison de la contrainte de positivité stricte. Dans l'interprétation des boucles, l'opérateur lfp (voir page 36) calcule une formulation logique du plus petit point fixe de son argument. Cet opérateur se codifie aisément comme suit.

```
Definition lfp { T U : Type } (F : @Specification T T' → @Specification T T')
: @Specification T T' :=
  fun s s' ⇒ ∀ X, (∀ s s', F X s s' → X s s') → X s s'.
```

L'interprétation des programmes sous la forme d'une logique à la Hoare que nous avons présentée dans la section 4.6.1 (page 42) pourrait aussi être considérée comme un élément d'interface puisque cette interprétation permet de donner une définition alternative et équivalente de la notion de raffinement. La codification de cette logique (figure 4.3 en page 43) se fait sans difficulté via le type inductif ci-dessous. Le type HTriple est un prédicat inductif paramétré par le type T des états initiaux, le type T' des états finaux, une précondition de type $T \to$ Prop, un programme de type Stmt $T\ T'$, et une postcondition de type $T' \to$ Prop. Chaque constructeur est la traduction d'une règle d'inférence du système de preuve de la figure 4.3.

```
Inductive HTriple { T } : ∀ T', (T → Prop) → @Stmt T T' → (T' → Prop) → Prop :=

| Assignment :
    ∀ { T' } (f : T → T') (P : T → Prop) (Q : T' → Prop),
      (∀ s, P s → Q (f s))
      → P {: (Assignment f) :} Q

| Seq :
    ∀ { Tx T' } P X Q (S1 : @Stmt T Tx) (S2 : @Stmt Tx T'),
    P {: S1 :} X → X {: S2 :} Q → P {: (Stmt.Seq S1 S2) :} Q

| If :
    ∀ { T' } (P : (T → Prop)) (Q : T' → Prop) (C : T → Prop)(St Sf : @Stmt T T'),
      (∀ s, P s → decidable (C s))
    → (fun s ⇒ C s ∧ P s) {: St :} Q
```

$\rightarrow$ (`fun` $s \Rightarrow \neg\ C\ s \wedge P\ s$) {: Sf :} Q
$\rightarrow P$ {: ($If\ C\ St\ Sf$) :} Q

| *While* :
$\forall\ W\ (P\ Q\ I\ C : T \rightarrow$ `Prop`$)\ (B :$ @*Stmt* $T\ T)\ (R : W \rightarrow W \rightarrow$ `Prop`$)\ (V : T \rightarrow W)$,
($\forall\ s, I\ s \rightarrow$ *decidable* $(C\ s)$)
$\rightarrow$ *well_founded* R
$\rightarrow (\forall\ s, P\ s \rightarrow I\ s)$
$\rightarrow (\forall\ v, ($`fun` $s \Rightarrow C\ s \wedge I\ s \wedge v = V\ s)$ {: B :} (`fun` $s \Rightarrow I\ s \wedge R\ (V\ s)\ v)$)
$\rightarrow (\forall\ s, \neg\ C\ s \rightarrow I\ s \rightarrow Q\ s)$
$\rightarrow P$ {: ($While\ C\ B$) :} Q

| *Spec* :
$\forall$ { T' } $(P : T \rightarrow$ `Prop`$)\ (Q : T' \rightarrow$ `Prop`$)\ ($*spec* : @*Specification* $T\ T')$,
$(\forall\ s\ , P\ s \rightarrow (\forall\ s', spec\ s\ s' \rightarrow Q\ s') \wedge (\exists\ s', spec\ s\ s'))$
$\rightarrow P$ {: ($Spec\ spec$) :} Q

`where` "P {: C :} Q" := (*HTriple* $P\ C\ Q$) (`at level` 50, *format* "P {: C :} Q").

Le schéma de traduction est systématique. Par exemple, le constructeur Seq est la traduction de la règle (seq) qui permet de construire le triplet valide P {: (Stmt.Seq $S_1\ S_2$) :} Q à partir des triplets P {: S_1 :} X et X {: S_2 :} Q. La relation entre pred et HTriple (voir Lemme 10 en page 45) se traduit en Coq par le lemme suivant.

`Lemma` *hoare_pred* :
$\forall\ T\ T'\ (C :$ @*Stmt* $T\ T')\ P\ Q, P$ {: C :} $Q \leftrightarrow P$ {: ($Stmt.Spec\ [\![C]\!]$) :} Q.

L'équivalence entre C et $[\![C]\!]$ décrite par le lemme hoare_pred fournit une règle d'élimination qui permet de passer d'une hypothèse ou d'un but de la forme $P\ \{C\}\ Q$ à un autre de la forme suivante :

$$\forall\ e \cdot P\ e \rightarrow\ (\exists\ e' \cdot [\![C]\!]\ e\ e')\ \wedge\ (\forall\ e' \cdot [\![C]\!]\ e\ e' \rightarrow Q\ e')$$

Pour ce faire, il suffit d'appliquer hoare_pred et ensuite de procéder par élimination du constructeur HTriple.Spec. Cette manœuvre est particulièrement utile dans la cas où C est de la forme $S_1;S_2$. En effet, en conséquence du fait que notre formalisation repose sur l'utilisation des types dépendants, une élimination directe d'une hypothèse de la

forme P {: (Stmt.Seq S_1 S_2) :} Q peut engendrer des hypothèses dont l'exploitation nécessite de supposer des axiomes tels que l'axiome K [37] ou celui d'*unicité des preuves d'égalité* (appelé *UIP*), qui permettent de considérer que toutes les preuves d'une égalité donnée sont des preuves égales. Cependant, reposer sur de tels axiomes rendrait notre mécanisation incompatible avec d'autres formalisations utilisant des axiomes incompatibles avec K ou *UIP* comme par exemple l'axiome d'univalence [78] qui permet de considérer que deux types isomorphes sont égaux. Le lemme hoare_pred permet de contourner cette difficulté en transformant l'hypothèse P' {: (Stmt.Seq S_1 S_2) :} Q' en l'hypothèse P' {: (Stmt.Spec $[\![S_1;S_2]\!]$) :} Q' dont l'élimination ne pose pas de difficultés. Par la même occasion ceci rend notre formalisation indépendante de K et de *UIP*. L'interprétation pred peut donc avoir une utilité pratique dans la mécanisation de résultats à propos de l'interprétation HTriple.

7.3.2 Relation de raffinement

Le troisième et dernier élément d'interface est la notion d'implémentation qui correspond à la définition relationnelle du raffinement de programmes (voir définition 14 en page 40).

```
Definition pred_refines { T T' : Type } (S1 S2 : @Stmt T T') :=
  ∀ s, (∃ s', ⟦S2⟧ s s') → (∀ s', ⟦S1⟧ s s' → ⟦S2⟧ s s') ∧ (∃ s', ⟦S1⟧ s s').
```

De même que l'interprétation HTriple peut être considérée comme un élément d'interface, la définition du raffinement basée sur cette interprétation (voir définition 16 en page 47) peut aussi être considéré comme tel. Dans le développement Coq, cette définition est représentée comme suit.

```
Definition hoare_refines { T T' : Type } (S1 S2 : @Stmt T T') :=
  ∀ P Q, P {: S2 :} Q → P {: S1 :} Q.
```

Notons que la définition hoare_refines est plus concise et peut sembler plus intuitive. Comme nous l'avons montré plus haut, cette définition est équivalente à hoare_pred (voir théorème 3 en page 49). Nous considérons donc le théorème ci-dessous qui établit cette équivalence comme faisant partie de l'interface externe de notre mécanisation.

```
Theorem hoare_refines_iff_pred_refines :
  ∀ T T' (Q R : @Stmt T T'), hoare_refines Q R ↔ pred_refines Q R.
```

7.3.3 Discussion

L'interprétation pred, qui décrit les transformations qui s'appliquent aux états individuels de programmes, est plus concise que l'interprétation HTriple, qui elle décrit les transformations qui s'appliquent à des ensembles d'états de programmes. De plus, l'interprétation pred peut sembler plus intuitive dans la mesure où en pratique l'exécution d'un programme ne transforme pas des ensembles d'états à chaque étape d'exécution, mais plutôt un seul état de programme à la fois. D'un autre côté la définition relationnelle du raffinement de programmes basée sur l'interprétation pred semble moins concise et moins intuitive que la définition utilisant l'interprétation HTriple. En effet, la seconde se lit : toute spécification satisfaite par S_2 est aussi satisfaite par S_1, et il semble difficile de faire plus simple. Vues comme des éléments d'interface, on peut donc dire que l'interprétation pred et l'interprétation HTriple sont complémentaires. Enfin, indépendamment des considérations d'interface, nous avons vu que l'interprétation pred peut constituer un tremplin utile pour mécaniser en pratique des résultats à propos de l'interprétation HTriple.

7.4 Développements et correction par construction

Le concept de *développement* évoqué dans les chapitres précédents permet de capturer tous les raffinements successifs qui jalonnent le passage d'une spécification à un programme concret. Ce concept ne peut donc pas échapper à notre mécanisation puisque cette dernière a pour objectif de permettre la validation de toutes les étapes de raffinement. Après avoir décrit la traduction en Coq du concept de développement, nous présentons l'encodage des contraintes imposées aux développements dans le but d'en garantir, par typage, la correction par construction. Enfin nous expliquerons comment, grâce à l'utilisation de la tactique Program, notre mécanisation s'intègre à l'assistant de preuve Coq afin de permettre à l'utilisateur de certifier ses développements.

7.4.1 Syntaxe et interprétation des développements

La syntaxe abstraite des développements (figure 6.2 en page 80) se traduit directement en le type inductif ci-dessous, chaque constructeur correspondant à une forme syntaxique donnée. Ce type inductif est paramétré par le type T de l'état d'initial, le type T' de l'état final, et le booléen b permettant de distinguer les développements contenants des blocs des développements qui n'en contiennent pas.

```
Inductive Dev { T } : ∀ (T' : Type)(b : bool), Type :=

| Spec : ∀ { T' }, @Specification T T' → @Dev T T' true

| Assignment : ∀ { T' }, (T → T') → @Dev T T' true

| Seq : ∀ { U T' b1 b2 }, @Dev T U b1 → @Dev U T' b2 → @Dev T T' (andb b1 b2)

| If : ∀ { T' b1 b2 } (p : T → Prop),
        @Dev T T' b1 → @Dev T T' b2 → @Dev T T' (andb b1 b2)

| While { b } (p : T → Prop) (d : @Dev T T b) : @Dev T T b

| Block : ∀ { T' b }, @Dev T T' true → @Dev T T' b → @Dev T T' false.
```

Le constructeur Spec a comme paramètre un objet de type @Specification T T'. Assignment est paramétré par un transformateur de type $T \to T'$. Seq est paramétré par deux développements de types respectifs Dev T U $b1$ et Dev U T' $b2$. If est paramétré par une condition de type $T \to$ Prop, ainsi que par deux développements de type Dev T T' b. while est paramétré par une condition de type $T \to$ Prop ainsi que par un corps de type Dev T T' b. Enfin, Block est paramétré par une abstraction de type Dev T T' true et par une concrétisation de type Dev T T' b. Comme nous l'avons expliqué précédemment, la définition de la correction des développements se réfère aux projections abstraites et concrètes de ces derniers. Ces projections (définition 21 en page 82) se concrétisent naturellement en Coq sous la forme de la fonction récursive suivante.

```
Fixpoint phi { T T' b } (D : @Dev T T' b): (@Stmt T T') × (@Stmt T T') :=
  match D with
  | Spec S1 ⇒ (Stmt.Spec S1, Stmt.Spec S1)
  | Assignment f ⇒ (Stmt.Assignment f, Stmt.Assignment f)
  | Seq d1 d2 ⇒ (Stmt.Seq (φa d1) (φa d2) , Stmt.Seq (φc d1) (φc d2) )
  | If p d1 d2 ⇒ (Stmt.If p (φa d1) (φa d2) , Stmt.If p (φc d1) (φc d2) )
  | While p d ⇒ (Stmt.While p (φa d) , Stmt.While p (φc d) )
  | Block s d ⇒ ( φa s , φc d )
  end
where "'φ' d" := (phi d) and "'φa' d" := (fst (phi d)) and "'φc' d" := (snd (phi d)).
```

La fonction phi permet de calculer à la demande une abstraction ou une concrétisation d'un développement D donné. Comme nous le verrons dans les prochaines sections, le calcul des abstractions intervient durant les raisonnements et permet de s'affranchir des détails d'implémentation. Et le calcul des concrétisations peut intervenir à la fin du processus de raffinement pour extraire le programme concret de type Stmt T T' qui aura été élaboré durant le développement.

7.4.2 Contraintes pour la correction par construction

Supposons que la notion de développement (section 6.3 en page 79) est représentée par un type inductif Dev T T' b, et que les projections φ_a et φ_c (définies dans la figure 21 en page 82) sont dérivées d'une fonction Coq φ : Dev T T' $\rightarrow$ Stmt T T'. Alors, les développements corrects par construction peuvent être représentées par les termes du type inductif ci-dessous.

```
Inductive CbC : ∀ { T T' b }, @Dev T T' b → Prop :=

| CbCSpec : ∀ { T T' } (S : @Specification T T'), CbC (Spec S)

| CbCAssignment : ∀ { T T' } (f : T → T'), CbC (Assignment f)

| CbCSeq : ∀ {T U T' b1 b2} { D1 : @Dev T U b1 } { D2 : @Dev U T' b2 },
             CbC D1 → CbC D2 → CbC (Seq D1 D2)

| CbCIf : ∀ { T T' b1 b2 } (p : T → Prop) { D1 : @Dev T T' b1 } { D2 : @Dev T T' b2 },
            CbC D1 → CbC D2 → CbC (If p D1 D2)

| CbCWhile : ∀ { T b } (p : T → Prop) {D : @Dev T T b},
               CbC D
               → well_founded (fun s s' ⇒ p s' ∧ pred (φa D)  s' s ∧ p s)
               → (let K := IIf p Then (φa D  End in K;K ⊑  K)
               → CbC (While p D)

| CbCBlock : ∀ { T T' b } {S1 : @Dev T T' true} {D1 : @Dev T T' b},
               CbC S1 → CbC D1 → (φa D1) ⊑ (φa S1)  → CbC (Block S1 D1).
```

Chacun des constructeurs du type CbC correspond à une règle d'inférence de la figure 6.3

(page 84) qui spécifie les règles de constructions des développements dits corrects par construction. Les constructeurs CbCSpec et CbCAssignment permettent de construire des fragments de développement correspondant à des spécifications ou des affectations. CbCSpec est paramétré par une spécification relationnelle, et CbCAssignment est paramétré par une transformation de T vers T'. Le constructeur CbcSeq permet de créer une séquence de deux développements D_1 et D_2. En plus de ces deux paramètres, ce constructeur exige les preuves que D_1 et D_2 sont des développements corrects par construction, d'où les paramètres supplémentaires CbC D_1 et CbC D_2. Le constructeur CbCBlock est celui qui permet de développer une abstraction en lui associant une concrétisation plus précise. Il est donc paramétré par une abstraction correcte par construction S_1 de type Dev T T' true et une concrétisation correcte par construction D_1 de type Dev T T' b. Afin de garantir la correction d'un bloc, le constructeur CbCBlock est également paramétré par une preuve que $\varphi_a\ D_1$ est bien un raffinement de $\varphi_a\ S_1 = \varphi_c\ S_1$ tel que le requiert la règle cbc-block de la figure 6.3. Dans le but d'améliorer la lisibilité des développements, nous définissons les notations suivantes.

```
Notation "' x := e"  := (CbCAssignment (fun x ⇒ e)) (at level 30, x pattern ...).
Notation "⟨ spec ⟩"  := (CbCSpec spec) (at level 0).
Notation "d1 ; d2"  := (CbCSeq  d1 d2  ) (at level 51, right associativity).
Notation "spec :{ impl }"  := (build_cd_block spec impl _) (at level 50).
```

Ces notations permettent d'encoder la première étape de raffinement de l'exemple décrit dans la section 6.2 (page 74) comme suit.

Exemple 17 (Premier raffinement de Sqrt)**.**

```
Definition sqr n := n × n.

Program Definition Sqrt :=
 ⟨  fun x '(x',r') ⇒ sqr r' ≤ x <  sqr (1 + r') ∧ x = x'  ⟩  :{
  'x := (x,0,x+1);
  ⟨  fun '(x,r,h) '(x',r',h') ⇒ sqr r ≤ sqr r' ≤ x ∧ x < sqr (S r') ≤ sqr h ∧ x = x' ⟩ ;
  '(x,r,h) := (x,r)
  }.
```

Dans le corps du bloc ci-dessus, la première instruction transforme un espace des états constitué d'une seule variable liée au nom x en un espace contenant trois variables respectivement initialisées aux valeurs x,0 et $x + 1$. Dans les instructions suivantes, les deux dernières variables initialisées à 0 et $x+1$ sont liées aux noms r et h. La variable h est en

fait une variable locale, et la dernière instruction du corps du bloc matérialise la sortie de h du champ lexical : l'espace des états à trois variables est transformé en un espace des états à deux variables.

À ce stade, le développement Sqrt n'est que partiellement acceptée par Coq. En effet, Coq est capable d'effectuer un typage partiel, mais la définition ne peut être acceptée en l'état puisqu'elle est incomplète. La partie manquante est le troisième paramètre de la fonction build_cbc_block dans la notation introduite pour les blocs. Dans cette notation, le paramètre manquant est un terme de preuve à partir duquel la fonction build_cbc_block est capable de tirer une preuve de correction du bloc, c'est-à-dire une preuve que $(\varphi_a\ D_1) \sqsubseteq (\varphi_a\ S_1)$, où D_1 est le corps du bloc et S_1 est l'abstraction du bloc. Comme nous l'avons expliqué précédemment, l'utilisateur gagnerait à ce qu'un énoncé brut de la forme $(\varphi_a\ D_1) \sqsubseteq (\varphi_a\ S_1)$ soit simplifié en utilisant par exemple un calcul de la plus faible pré-spécification. Ceci est la raison de l'introduction de la fonction build_cbc_block dont la tâche consiste à exiger un terme de preuve pour un énoncé simplifié, et ensuite à transformer ce terme en un habitant de $(\varphi_a\ D_1) \sqsubseteq (\varphi_a\ S_1)$. De manière analogue aux notations pour la séquence et les blocs, on définit respectivement les notations pour la sélection et les boucles comme suit.

```
Notation "'If' p 'Then' d1 'Else' d2 'End'" := (CDIf p d1 d2) (at level 90).
Notation "'While' p 'Do' d 'Done'" := (build_cbc_while p d _ _) (at level 50).
```

La fonction build_cbc_while sert à construire un développement de boucle correct à partir d'une condition d'arrêt, d'un corps de boucle, et de deux preuves permettant de garantir que les conditions d'abréviation de boucle (voir lemme 16 en page 66) sont satisfaites. Le catalogue des notations est maintenant suffisant pour encoder le développement de la figure 7.1 ci-dessous. Il est vrai que de devoir lier les noms des variables à chaque instruction n'est pas très pratique, cependant le résultat reste lisible.

En laissant des paramètres de build_cbc_block et build_cbc_while insuffisamment spécifiés dans les notations pour les blocs et les boucles, on crée des trous que l'on demande à Coq de combler par inférence. En général, l'inférence ne pourra pas réussir puisqu'il s'agit de termes de preuves pour des types qui sont trop généraux. Malgré cela, l'utilisation de la tactique **Program** nous permet de faire accepter cette définition partiellement à Coq. Le développement peut donc continuer à être élaboré par exemple en donnant une implémentation plus précise à des spécifications abstraites, ou tout simplement en changeant de stratégie de raffinement. Néanmoins, le système retient que le travail n'est pas achevé et qu'il reste des preuves à fournir.

```
Program Definition Sqrt :=
  ⟨ (fun x '(x',r') ⇒ sqr r' ≤ x < sqr (1 + r') ∧ x = x') ⟩ :{
    'x := (x,0,x+1);
    ⟨ fun '(x,r,h) '(x',r',h') ⇒ (sqr r ≤ sqr r' ≤ x ∧ x < sqr (S r') ≤ sqr h) ∧ x = x' ⟩ :{
        While (fun '(x,r,h) ⇒ 1 + r ≠ h) Do
          ⟨ fun '(x,r,h) '(x',r',h') ⇒ (sqr r ≤ sqr r' ≤ x ∧ x < sqr h' ≤ sqr h)
                                       ∧ x = x' ∧ (r,h) ≠ (r',h') ⟩ :{
            ⟨ fun '(x,r,h) '(x',r',h',m') ⇒ r < m' < h ∧ x = x' ∧ r = r' ∧ h = h' ⟩ :{
              '(x,r,h) := (x,r,h,r + Nat.div2 (h - r))
            };
            If (fun '(x,r,h,m) ⇒ sqr m ≤ x) Then
              '(x,r,h,m) := (x,m,h)
            Else
              '(x,r,h,m) := (x,r,m)
            End
        }
        Done
    };
    '(x,r,h) := (x,r)
  }.
```

Figure 7.1: Le développement Sqrt formalisé en Coq

7.4.3 Certification des développements

Afin de compléter un développement en cours ou déjà achevé, l'utilisateur peut à tout moment se donner la possibilité de fournir les preuves manquantes en exécutant la tactique **Next Obligation**. À chaque exécution de cette commande Coq demande à l'utilisateur de fournir une des preuves manquantes. Il faudra donc d'exécuter cette commande successivement et autant de fois que nécessaire pour remplir toutes les obligations de preuve. À l'exception des boucles et des blocs, les développements peuvent être librement composés entre eux, et la correction de l'ensemble découlera directement de la transitivité de la relation de raffinement, ou de la monotonie des structures de contrôle vis-à-vis de cette relation. En revanche, la création des blocs ou des boucles engendre des obligations de preuve.

Obligations de preuve engendrées par les blocs. Dans le cas des blocs, la preuve demandée correspond au paramètre proof de la fonction build_cbc_block ci dessous.

```
Definition build_cbc_block{ T U b} { spec : @Dev T U true } { impl }
  (cdspec : @CbC T U true spec) (cdimpl : @CbC T U b impl)
  (proof : let S := (φ_a spec) in (∀ s, (∃ s', pred S s s') → wpr' impl (pred S) s s))
  := CbCBlock cdspec cdimpl (refines_if_wpr' _ cdimpl proof).
```

La fonction build_cbc_block construit un bloc correct par construction à partir de la spécification du bloc spec, du corps du bloc impl, et de la preuve fournie par l'utilisateur proof. Le type de proof est une obligation de preuve calculée via un calcul de la plus faible pré-spécification (voir section 5.3). Le calcul de l'obligation de preuve fait appel à la fonction wpr' ci-dessous.

```
Fixpoint wpr' { T U T' : Type } { b } (D : @Dev U T' b) (S : @Specification T T')
: @Specification T U :=
  match D with
  | Spec S1 ⇒ fun S s s' ⇒ (∀ sx, S1 s' sx → S s sx) ∧ (∃ sx, S1 s' sx)
  | Assignment f ⇒ fun S s s' ⇒ S s (f s')
  | Seq C1 C2 ⇒ fun S ⇒ wpr' C1 (wpr' C2 S)
  | If p Ct Cf ⇒ fun S s s' ⇒ p s' ∧ wpr' Ct S s s' ∨ ¬ p s' ∧ wpr' Cf S s s'
  | While p C ⇒
    fun S s s' ⇒ p s' ∧ wpr' C (fun s s' ⇒ ¬ p s' → S s s') s s' ∨ ¬ p s' ∧ S s s'
  | Block S1 D1 ⇒ fun S ⇒ wpr' S1 S
  end S.
```

La fonction wpr' est un transformateur de spécification analogue au transformateur wpr (définition 20 en page 62). Comme wpr' s'applique dans un contexte dans lequel la construction des boucles requiert que les conditions d'abréviation soient satisfaites, sa définition est différente de celle de wpr dans le cas des boucles. En effet, l'obligation de preuve calculée pour le cas des boucles prend en compte le théorème 7 (page 85) qui permet d'exploiter le cas échéant la structure du corps de la boucle. Pour construire un block correct, la fonction build_cbc_block fait appel au constructeur CbCBlock. Comme ce dernier attend une preuve de $(\varphi_a\ impl) \sqsubseteq (\varphi_a\ spec)$, la preuve fournie par l'utilisateur doit être convertie en conséquence. Pour ce faire, build_cbc_block applique le théorème refines_if_wpr' suivant.

```
Theorem refines_if_wpr' :
  ∀ {T T' b} {D : @Dev T T' b} S,
    CbC D → (∀ s, (∃ s', pred S s s') → wpr' D (pred S) s s) → (φa D) ⊑ S  .
```

L'application du théorème refines_if_wpr' est une indirection qui ne remet pas en cause les résultats de correction (théorème 8) et complétude (théorème 9) exposés dans la sec-

tion 6.4. En effet, le théorème wpr_iff_wpr' ci-dessous établit une équivalence entre wpr et wpr' dans un contexte de correction par construction (hypothèse (CbC D)), or cette hypothèse est bien vraie à chaque fois que l'indirection a lieu.

```
Theorem wpr_iff_wpr' : ∀ {T U T' b} {D : @Dev U T' b},
    CbC D → ∀ (S : @Specification T T') s s', wpr (φ_a D) S s s' ↔ wpr' D S s s'.
```

À la suite du premier raffinement présenté dans l'exemple 17, l'exécution de la commande Next Obligation place l'utilisateur dans le contexte de preuve suivant.

```
============================================================
∀ s : nat,
(∃ '(x', r'), sqr r' ≤ s < sqr (S r') ∧ s = x') →
wpr_spec (fun '(x, r, h) '(x', r', _) ⇒ (sqr r ≤ sqr r' ≤ x ∧ x < sqr (S r') ≤ sqr h) ∧ x = x')
         (fun (s0 : nat) (s' : nat × nat × nat) ⇒
            let '(x', r') := let '(x, r, _) := s' in (x, r) in sqr r' ≤ s0 < sqr (S r') ∧ s0 = x')
         s (s, 0, s + 1)
```

La fonction wpr_spec référencée dans le but à établir est un relèvement du transformateur wpr au niveau des spécifications comme indiqué ci-dessous. Cette fonction est la traduction en Coq de la définition 19 (page 61) de la plus faible pré-spécification.

```
Definition wpr_spec { T U T' } (C : @Specification U T') (S : @Specification T T')
: @Specification T U := fun s s' ⇒ (∀ sx, C s' sx → S s sx) ∧ (∃ sx, C s' sx).
```

Il s'agit ensuite de décharger le but présenté par le système en utilisant les tactiques offertes par le système Coq. Dans le cas qui nous concerne, l'obligation de preuve peut être déchargée grâce aux commandes suivantes.

```
Next Obligation.
  intros s ((x',r'),(HH,HHx)); subst x'; simpl in *.
  split; try (intros ((xx,rx),hx); intros ((HHx,HHx'),HHprog); subst); try (∃ ((s,r'),S r')).
  { unfold sqr in *; simpl; split; auto. nia. }
  { split; auto.
    { unfold sqr in *. split.
      { nia. }
      { split.
        { nia. }
        { assert (r' ≤ s) by nia; clear HH; nia. }}}}
Qed.
```

L'obligation de preuve n'est pas très engageante au départ, mais on peut assez rapidement se ramener aux termes du problème à résoudre en utilisant les tactiques d'introduction comme intros afin de nommer les variables et les hypothèses. Ensuite, la tactique nia, qui invoque une procédure de décision (incomplète) pour l'arithmétique non linéaire, nous permet de conclure. Dès l'exécution des trois premières lignes de commandes, le contexte de preuve qui contient deux buts est déjà beaucoup plus abordable comme on peut le voir ci-dessous.

r', xx : nat
HH : $sqr\ r' \leq xx < \ sqr\ (S\ r')$
rx, hx : nat
HHx' : $xx < \ sqr\ (S\ rx) \leq sqr\ (xx + 1)$
HHx : $sqr\ 0 \leq sqr\ rx \leq xx$
==========
$sqr\ rx \leq xx < \ sqr\ (S\ rx) \wedge xx = xx$

s, r' : nat
HH : $sqr\ r' \leq s < \ sqr\ (S\ r')$
==========
$(sqr\ 0 \leq sqr\ r' \leq s \wedge s < \ sqr\ (S\ r') \leq sqr\ (s + 1)) \wedge s = s$

Obligations de preuve engendrées par les boucles. Précédemment, nous avons introduit une notation permettant de faciliter l'encodage d'un développement de boucle. Cette notation cache un appel de la fonction build_cbc_while avec deux trous. La définition en Coq de cette fonction est indiquée ci-dessous.

```
Definition build_cbc_while
  { T b} { d : @Dev T T b }
  (p : @Predicate T)
  (cd : @CbC T T b d)
  (proof_wf : well_founded (fun s s' ⇒ p s' ∧ pred (φa d)  s' s ∧ p s))
  (proof_body : (let K := If p d (Assignment (fun s ⇒ s)) in
                 (∀ s, (∃ s', pred (φa K)  s s') → wpr' (Seq K K) (pred (φa K)  )s s)))
  := let CbCK := CDIf p cd (CDAssignment (fun s ⇒ s)) in
   CDWhile p cd proof_wf (refines_if_wpr' _ (CbCSeq CbCK CbCK) proof_body).
```

Les deux trous dans la notation correspondent aux paramètres proof_wf et proof_body qui sont des preuves à fournir permettant de garantir que le développement de boucle rempli les critères imposés par les règles de construction. Le premier paramètre attendu (proof_wf) correspond à une preuve que l'abstraction du corps de la boucle, est une relation bien fondée lorsque son domaine et son codomaine sont restreints aux états de programmes qui satisfont la condition de la boucle. Le second paramètre attendu (proof_body) est une preuve qu'une séquence de deux pas d'exécution de la boucle ($K;K$) raffine un seul pas d'exécution (K), sachant que, encore une fois, on se contente de raisonner avec des abstractions dont la pertinence devra être établie par ailleurs. Pour décharger ces obligations de preuve, la démarche interactive à suivre est la même que celle évoquée précédemment dans le cas des obligations de preuve engendrées par les blocs.

Exhaustivité de la vérification. L'exhaustivité de la vérification peut aisément être vérifiée. En effet, il suffit d'utiliser la commande Check pour voir si notre développement est un objet référençable dans le système. Par exemple, la commande Check Sqrt. échouera si et seulement si le développement Sqrt contient des trous, c'est-à-dire si et seulement si ce développement a généré des obligations de preuve qui n'ont pas été déchargées. Lorsque la vérification est exhaustive, il est possible de produire un certificat de correction. Pour ce faire, il suffit d'appliquer le théorème de correction principal de notre mécanisation (théorème 8). L'énoncé Coq correspondant à ce théorème est le suivant.

`Theorem` *soundness* : $\forall\ \{T\ \ T'\ \ b\}\ \{D : @Dev\ \ T\ \ T'\ \ b\},\ CbC\ \ D \rightarrow (\varphi_c\ \ D) \sqsubseteq (\varphi_a\ \ D)$.

Les notations utilisées pour construire un développement permettent de produire un terme de type (CbC D). Par exemple l'objet Sqrt mentionné plus haut est de type (CbC D) dès que toutes les obligations de preuves sont déchargées. On peut alors, en exécutant la commande ci-dessous, obtenir une preuve de correction de ce développement, c'est-à-dire une preuve que $(\varphi_c\ \mathsf{Sqrt}) \sqsubseteq (\varphi_a\ \mathsf{Sqrt})$.

`Definition` *Sqrt_proof* : $(\varphi_c\ \mathsf{Sqrt}) \sqsubseteq (\varphi_a\ \mathsf{Sqrt})\ :=$ *soundness Sqrt.*

On remarquera que l'utilisateur n'a pas à invoquer explicitement les propriétés de la relation de raffinement telles que la transitivité, ou encore la monotonie vis-à-vis des structures de contrôle. En fait, l'application du théorème soundness afin de produire un certificat revient implicitement à appliquer ces propriétés de manière transparente. La propriété de transitivité s'applique à chaque fois que des blocs sont imbriqués, la monotonie

de (;) vis-à-vis de la relation de raffinement s'applique à chaque fois que les développements sont composés en séquence, et il en est de même pour les autres structures de contrôle.

7.5 Travaux connexes

Dans [4], Alpuim et Swierstra proposent une librairie *Coq* de *predicate transformers* permettant de développer par raffinements des programmes impératifs dans le langage *While*. La librairie est basée sur un plongement léger en *Coq* du *refinement calculus* limité à la correction partielle. Elle permet d'exprimer les spécifications en associant une précondition et une postcondition. La sémantique associée à une spécification définie ainsi est la plus faible précondition permettant de satisfaire cette spécification. À la manière de Morgan, une instruction spécifique permettant de formaliser les spécifications est ajouté au langage. Le raffinement entre deux spécifications est alors défini par l'affaiblissement de leur préconditions respectives et le renforcement de leurs postconditions respectives. Pour pouvoir exprimer la relation de raffinement entre deux programmes, ces derniers sont traduits en spécifications. Cette traduction se fait par induction sur la syntaxe. Ensuite la relation de raffinement entre programmes est réduite à la relation de raffinement entre les spécifications correspondantes. Ce plongement des concepts du *refinement calculus* permet d'en prouver les règles de dérivation dans *Coq*. La librairie propose donc une interface composée des tactiques permettant d'appliquer les règles du *refinement calculus* afin de raffiner une spécification jusqu'à obtenir un programme exécutable, c'est-à-dire un programme ne faisant pas usage de l'instruction de spécification.

Dans [13], Boulmé présente un plongement du *refinement calculus* de Morgan dans Coq. La représentation donnée aux programmes correspond à une monade M, et les spécifications sont représentées par une extension de M. Les conditions de raffinement sont calculées et simplifiées grâce à un calcul de la plus faible précondition. Cette formalisation considère la correction partielle et la correction totale. La sémantique des boucles est formalisée selon une théorie de point fixe dans le treillis des spécifications muni de la relation de raffinement. Bien que la formalisation inclut les résultats correspondant aux règles de raffinement que l'on peut trouver dans le calcul de Morgan, il revient au programmeur de les appliquer explicitement lors des preuves de raffinements.

7.6 Conclusion

Dans ce chapitre nous avons présenté une mécanisation [2], en environ 4600 lignes de script Coq, d'une théorie du raffinement de programme basée sur les approches relationnelles et exprimée dans le langage de la théorie des types. L'interface de la mécanisation est constituée de la syntaxe des programmes, de la sémantique prédicative associée, ainsi que de la définition de la relation de raffinement. Nous avons aussi mécanisé une sémantique à la Hoare. Cette sémantique alternative et logiquement équivalente, est complémentaire à la sémantique relationnelle et peut aussi servir d'interface à notre mécanisation. Ces éléments d'interface sont les seuls auxquels l'utilisateur de la mécanisation doit accorder sa confiance. Nous avons intégré notre mécanisation dans l'infrastructure de l'assistant de preuve Coq notamment par l'utilisation de la tactique Program qui permet d'élaborer des termes de manière interactive. Notre mécanisation se différencie des travaux de Alpuim et Swierstra [4] par le fait qu'elle ne se limite pas à la correction partielle mais prend en compte la correction totale des développements. De plus, notre approche améliore la lisibilité des développements par l'utilisation du concept de bloc spécifié et par l'utilisation de la tactique Program qui permet de séparer les instructions de développement des scripts de preuve. Comparée aux travaux de Boulmé [13], notre mécanisation utilise une théorie relationnelle plutôt qu'une théorie basée sur les transformateurs de prédicats de Dijkstra. La mécanisation s'en trouve simplifiée car, par exemple, la théorie du point fixe que nous avons encodée s'exprime dans le cadre du treillis des relations ordonnées par l'inclusion ensembliste, alors que la théorie du point fixe utilisée par Boulmé s'exprime dans le cadre du treillis des spécifications pré/post ordonnées par la relation plus complexe de raffinement. Enfin, là où les travaux connexes mentionnés se limitent à la correction de la théorie, nous avons pris la peine d'établir formellement et de mécaniser la complétude de notre théorie du raffinement de programmes.

[2] https://github.com/bsall/thesis-dev

Chapitre 8

Conclusion & Perspectives

8.1 Conclusion

Il est raisonnable de penser qu'on doit pouvoir établir la correction d'un programme vis-à-vis d'une spécification, cependant il est certain qu'une preuve de correction vis-à-vis d'une spécification erronée n'a aucune valeur. Établir la correction d'une spécification n'a pas de sens, une spécification doit être suffisamment intelligible pour pouvoir servir de contrat entre celui qui spécifie et celui qui programme. Pour réduire le risque d'erreur dans les spécifications, une méthode consiste à élever le niveau d'abstraction des spécifications en les exprimant dans un langage de haut niveau, car en réduisant le niveau de détail, on réduit aussi les opportunités d'erreurs. De plus, certains outils de haut niveau, tel que les assistants de preuves, peuvent alors être mis à profit pour raisonner sur les modèles afin d'en prouver la correction ou d'en découvrir les erreurs. Néanmoins, entre une spécification intelligible de haut niveau et un programme efficace, il peut exister un écart important, donc difficile à combler en une seule fois. D'où l'intérêt de procéder par étapes de raffinements successifs. En validant chaque étape de raffinement avant de passer à la suivante, on s'assure que le programme final satisfait par transitivité la spécification initiale. Ces validations nécessitent un effort de preuve important, mais qui se justifie lorsque la correction des erreurs induit des coûts encore plus importants. Ceci est en particulier le cas dans certains systèmes critiques.

Dans cette thèse, nous avons formalisé une théorie de la programmation par raffinements. La formalisation se base sur le paradigme de la programmation prédicative qui a été initié par Hehner, et qui donne une vision relationnelle et prédicative de la sémantique des programmes. Alors que les formalisations classiques basées sur la logique de Hoare décrivent à la fois la sémantique des programmes et le système de preuve, les approches relationnelles permettent de décrire la sémantique des programmes indépendamment du

système de preuve. De plus, les approches relationnelles présentent l'intérêt de traiter les programmes et les spécifications de manière uniforme. D'une part, ceci permet de simplifier la formulation de la théorie puisque la sémantique des boucles dans un cadre non-déterministe est complètement définie par un plus petit point fixe. D'autre part, les spécifications relationnelles permettent de capturer le comportement des boucles avec plus de flexibilité que les invariants, et de la même manière que tout autre fragment de programme. Et enfin, les approches relationnelles qui décrivent les programmes comme des relations entre états de programme sont plus proches du comportement réel des programmes à l'exécution que les approches plus classiques qui décrivent les programmes comme des relations entre ensembles d'états de programme. En effet, l'exécution des programmes procède en général de la transformation d'un seul état par étape d'exécution plutôt que de la transformation de plusieurs états à la fois. Une interprétation proche du comportement effectif des programmes à l'exécution est désirable pour la simple et bonne raison que toute preuve de correction s'entend vis-à-vis d'une interprétation particulière des programmes sur laquelle les parties prenantes doivent s'accorder. Par conséquent, le fait qu'une telle interprétation soit proche de la réalité opérationnelle en se détachant du problème de la preuve, aide à élever le niveau global de confiance dans la mesure où cela facilite la communication entre par exemple celui qui écrit une spécification et celui qui en développe une implémentation, ou encore entre celui qui spécifie la sémantique d'un langage et celui qui en implémente le compilateur.

Lorsqu'il s'agit de prouver formellement la correction d'une implémentation vis-à-vis d'une spécification, l'utilisation d'une logique de programme dédiée à cela est préférable à la preuve directe basée sur l'interprétation des programmes dans le cadre relationnel. La difficulté provient de l'opérateur de composition séquentiel et des boucles. Classiquement, ce problème est résolu par le calcul de la plus faible précondition, et par l'invention d'invariants pour capturer une abstraction de l'ensemble des états accessibles pendant l'exécution d'une boucle. L'approche que nous avons choisie, s'appuie sur un calcul de la plus faible pré-spécification qui est un analogue du calcul de la plus faible précondition ayant la même puissance que cette dernière, mais plus adapté au cadre relationnel parce que manipulant une seule notion de spécification et restant ainsi dans la continuité d'un traitement uniforme des programmes et des spécifications. Dans le cas des boucles, les approches classiques s'appuient généralement sur une mesure décroissante pour traiter le problème de la terminaison. Cette approche cache une démarche relationnelle puisque la relation de décroissance associe les deux valeurs résultant de l'application d'une même fonction à un état initial et à un état final. Ceci avait conduit Manna et Pnueli à une approche plus relationnelle de la correction des boucles limitée au cas

déterministe [48], et dont une généralisation a plus tard été prouvée complète dans [20]. Le cas non-déterministe est considéré dans la démarche de Frappier [30] mais cette dernière dévie quelque peu de la démarche initiale de Manna et Pnueli de sorte qu'elle est incomplète. En cherchant à généraliser la démarche de Frappier, nous avons aussi obtenu une généralisation de la méthode de Manna et Pnueli au cas non-déterministe. En outre, nous montrons que notre généralisation est complète.

Le développement des programmes non triviaux est difficilement réalisable d'une traite. En général plusieurs étapes se succèdent, et durant ces étapes le programmeur fait des choix d'implémentation qui vont déterminer l'adéquation et la performance des programmes finaux. Chacun des choix qui jalonnent la construction d'un programme est aussi une opportunité d'erreur. Ainsi, dans une démarche de vérification a posteriori, plusieurs erreurs auront pû être accumulées avant que la vérification n'intervienne. Ceci ne facilite pas les choses puisqu'il faut alors se replacer dans le contexte précis dans lequel l'erreur a été commise pour pouvoir l'identifier et la corriger avec le risque que cela peut produire des effets de bord indésirables. Une démarche de vérification a priori permet de déceler les erreurs au plus près du contexte même dans lequel elles sont commises. De plus, si la vérification est effectuée étape par étape au fur et à mesure pendant la construction du programme, alors la complexité est plus facile à maîtriser. Ces constatations sont à l'origine de la méthode de développement par raffinements qui a été initialement formulée par Wirth et Dijkstra. Pour être applicable en pratique, le développement par raffinements nécessite néanmoins des outils permettant une mise en œuvre fidèle des principes théoriques. La formalisation que nous avons effectuée s'inscrit dans cet objectif. Elle est basée sur une théorie formellement prouvée, et mécanisée dans l'assistant de preuve Coq. Ce qui permet d'obtenir des développements certifiés dont la conformité aux spécifications est mécaniquement vérifiable, envers une base de confiance qu'il est humainement possible d'auditer.

8.2 Perspectives

La démarche de raffinement peut s'arrêter dès que l'on atteint un programme suffisamment concret, c'est-à-dire, un programme dont toutes les instructions peuvent être transcrites dans un langage de programmation avant d'être compilées et exécutées. Cependant, il existe encore un risque d'introduction d'erreurs durant ces ultimes étapes de traduction et de compilation. Concernant la phase de compilation, la dernière décennie a vu naître une nouvelle génération de compilateurs optimisants certifiés corrects sur lesquelles nous pouvons nous appuyer. Parmi ces compilateurs, on peut citer *CompCert* [46] pour le

langage C, *CakeML* [44] et *Pilsner* [60] pour *ML*, ou encore *CertiCoq* [5] pour *Gallina*, le langage de programmation de Coq.

Une direction naturelle de la poursuite de nos travaux pourrait donc consister à couvrir la phase de traduction en automatisant la production de code pour ces compilateurs à partir des développements suffisamment concrets. La programmation d'un tel traducteur peut se faire au sein même de Coq en utilisant par exemple la librairie *MetaCoq* [72] qui permet de réifier les termes de Coq dans son propre langage de programmation. Néanmoins, il est nécessaire de passer par une phase d'implémentation qui est la phase de raffinement dont l'objectif est de tenir compte des limites physiques telles que par exemple la taille finie des nombres que les machines peuvent traiter. En outre, il faut aussi s'assurer de la décidabilité des conditions associées aux instructions de branchement. À l'occasion d'une étape d'implémentation, les conditions des instructions de branchement doivent être remplacées par des expressions booléennes, et plus généralement, toutes les expressions associées aux conditions de branchement ou aux affectations doivent faire appel à des calculs qui reflètent fidèlement les possibilités de la machine cible. Par exemple, les opérations arithmétiques exécutées par la machine cible peuvent correspondre à une arithmétique modulaire, et dans ce cas, l'arithmétique non bornée utilisée dans les expressions doit être remplacée par une arithmétique modulaire.

Un autre axe de poursuite de nos travaux concerne l'enrichissement du langage de programmation que nous avons étudié. En effet, pour pouvoir envisager des développements à plus grande échelle, il faut une notion de procédure. De plus, une notion plus explicite de variable, ainsi que le support de la récursion faciliteraient grandement l'applicabilité en pratique de la théorie. Enfin, afin de permettre l'usage efficace de certaines structures de données, une logique de séparation [67] permettant le raisonnement sur le tas et les pointeurs semble incontournable.

Bibliographie

[1] Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. All for the Price of Few. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013*, volume 7737 of *LNCS*, pages 476–495. Springer, 2013.

[2] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[3] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466, 2010.

[4] João Alpuim and Wouter Swierstra. Embedding the refinement calculus in Coq. *Science of Computer Programming*, 164:37–48, 2018.

[5] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: A verified compiler for coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

[6] Ralph-Johan Back. *On the correctness of refinement in program development*. PhD thesis, Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

[7] Ralph-Johan Back. A Calculus of Refinements for Program Derivations. *Acta Informatica*, 25(6):593–624, 1988.

[8] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[9] Roland C. Backhouse and Jaap van der Woude. Demonic Operators and Monotype Factors. *Mathematical Structures in Computer Science*, 3(4):417–433, 1993.

[10] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[11] Rudolf Berghammer and Hans Zierer. Relational Algebraic Semantics of Deterministic and Nondeterministic programs. *Theoretical Computer Science*, 43:123–147, 1986.

[12] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[13] Sylvain Boulmé. Intuitionistic Refinement Calculus. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007*, volume 4583 of *LNCS*, pages 54–69. Springer, 2007.

[14] Michael Butler, Jim Grundy, Thomas Langbacka, Rimvydas Ruksenas, and Joakim von Wright. The Refinement Calculator: Proof Support for Program Refinement. In *Formal Methods Pacific '97*, pages 40–61. Springer, 1997.

[15] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[16] Edmund M. Clarke, Jr., Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. The Cyber-Physical Systems Series. MIT Press, Cambridge, MA, second edition, 2018.

[17] Stephen A. Cook. Soundness and Completeness of an Axiom System for Program Verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[18] Jack Copeland. *Colossus: The secrets of Bletchley Park's code-breaking computers*. Oxford Universisty Press, 2010.

[19] Thierry Coquand. *Une théorie des constructions*. PhD thesis, Université Paris Diderot, France, 1985.

[20] Patrick Cousot. On fixpoint/iteration/variant induction principles for proving total correctness of programs with denotational semantics. In Maurizio Gabbrielli, editor, *Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019*, volume 12042 of *LNCS*, pages 3–18. Springer, 2019.

[21] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, POPL 77*, pages 238–252. ACM, 1977.

[22] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In Shmuel Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[23] Jules Desharnais, Ali Jaoua, Fatma Mili, Noureddine Boudriga, and Ali Mili. A Relation Division Operator: The Conjugate Kernel. *Theoretical Computer Science*, 114(2):247–272, 1993.

[24] Edsger W. Dijkstra. Notes on structured programming, 1970.

[25] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.

[26] ClearSy System Engineering. Atelier, B. https://www.atelierb.eu/en/atelier-b-tools/.

[27] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods In System Design*, 48(3):152–174, 2016.

[28] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - Where Programs Meet Provers. In Matthias Felleisen and Philippa Gardner, editors, *European Symposium on Programming, ESOP 2013*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[29] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium in Applied Mathematics, American Mathematical Society*, pages 19–32. 1967.

[30] Marc Frappier, Ali Mili, and Jules Desharnais. A Relational Calculus for Program Construction by Parts. *Science of Computer Programming*, 26(1-3):237–254, 1996.

[31] Eric C. R. Hehner. Specifications, Programs, and Total Correctness. *Science of Computer Programming*, 34(3):191–205, 1999.

[32] Eric C. R. Hehner. Specified blocks. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005*, volume 4171 of *LNCS*, pages 384–391. Springer, 2005.

[33] Eric C. R. Hehner. *A practical theory of programming*. Springer Science & Business Media, 2012.

[34] C. A. R. Hoare and He Jifeng. *Unifying theories of programming*, volume 14. Prentice Hall, 1998.

[35] Charles A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[36] Charles A. R. Hoare and Jifeng He. The Weakest Prespecification. *Inf. Process. Lett.*, 24(2):127–132, 1987.

[37] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. *Twenty-five years of constructive type theory (Venice, 1995)*, 36:83–111, 1998.

[38] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[39] Éric Jaeger and Catherine Dubois. Why Would You Trust B ? In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007*, volume 4790 of *LNCS*, pages 288–302. Springer, 2007.

[40] Clifford B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[41] Mark B. Josephs. An introduction to the theory of specification and refinement. In *IBM research Report RC 12993*. IBM Thomas J. Watson Research Division, 1987.

[42] Donald E. Knuth. Literate Programming. *Comput. J.*, 27(2):97–111, 1984.

[43] Derrick G. Kourie and Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, 2012.

[44] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014*, pages 179–192. ACM, 2014.

[45] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[46] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[47] Richard C. Linger, Harlan D. Mills, and Bernard I. Witt. Structured programming - theory and practice. 1979.

[48] Zohar Manna and Amir Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 3:243–263, 1974.

[49] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.

[50] Ali Mili. A Relational Approach to the Design of Deterministic Programs. *Acta Informatica*, 20:315–328, 1983.

[51] Ali Mili, Jules Desharnais, and Fatma Mili. Relational Heuristics for the Design of Deterministic Programs. *Acta Informatica*, 24(3):239–276, 1987.

[52] Harlan D. Mills. The New Math of Computer Programming. *Communications of the ACM*, 18(1):43–48, 1975.

[53] Harlan D. Mills, R Austing, Victor R. Basili, John D. Gannon, Richard G. Hamlet, J. Kohl, and B. Schneiderman. The Calculus of Computer Programming. *Ally and Bacon, Inc.*, 1982.

[54] Harlan D. Mills, Victor R. Basili, John D. Gannon, and Richard G. Hamlet. Principles of Computer Programming: A Mathematical Approach. *Allyn and Bacon, Inc.*, 1987.

[55] Lee Momtahan. Towards a Small Model Theorem for Data Independent Systems in Alloy. *Electronic Notes Theoretical Computer Science*, 128(6):37–52, 2005.

[56] Carroll Morgan. The Specification Statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.

[57] Carroll Morgan. The Refinement Calculus. In Manfred Broy, editor, *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, 1992*, volume 118 of *NATO ASI Series*, pages 3–52. Springer, 1992.

[58] Carroll Morgan. The Refinement Calculus, and Literate Development. In Bernhard Möller, Helmuth Partsch, and Stephen A. Schuman, editors, *Formal Program Development - IFIP TC2/WG 2.1 State-of-the-Art Report*, volume 755 of *LNCS*, pages 161–182. Springer, 1993.

[59] Carroll Morgan. *Programming from specifications, (2nd ed.)*. Prentice Hall International series in computer science. Prentice Hall, 1994.

[60] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 166–178. ACM, 2015.

[61] Hanne R. Nielson and Flemming Nielson. *Semantics with applications - a formal introduction*. Wiley professional computing. Wiley, 1992.

[62] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[63] Catherine Parent. Synthesizing Proofs from Programs in the Calculus of Inductive Constructions. In Bernhard Möller, editor, *Mathematics of Program Construction, MPC'95*, volume 947 of *LNCS*, pages 351–379. Springer, 1995.

[64] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions. (Program Extraction in the Calculus of Constructions)*. PhD thesis, Université Paris Diderot, France, 1989.

[65] Christine Paulin-Mohring. Introduction to the coq proof-assistant for practical software verification. In *LASER Summer School on Software Engineering*, pages 45–95. Springer, 2011.

[66] Marie-Laure Potet. Spécifications et développements structurés dans la méthode B. *Technique et Science Informatiques*, 22(1):61–88, 2003.

[67] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.

[68] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick G. Kourie, and Bruce W. Watson. Tool Support for Correctness-by-Construction. In Reiner Hähnle and Wil M. P. van der Aalst, editors, *Fundamental Approaches to Software Engineering, 22nd International Conference, FASE 2019*, volume 11424 of *LNCS*, pages 25–42. Springer, 2019.

[69] Boubacar Demba Sall, Frédéric Peschanski, and Emmanuel Chailloux. A mechanized theory of program refinement. In Yamine Aït Ameur and Shengchao Qin, editors, *Formal Methods and Software Engineering - 21st International Conference on Formal Engineering Methods, ICFEM 2019*, volume 11852 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2019.

[70] Emil Sekerinski. A Calculus for Predicative Programming. In Richard S. Bird, Carroll Morgan, and Jim Woodcock, editors, *Mathematics of Program Construction, Second International Conference*, volume 669 of *LNCS*, pages 302–322. Springer, 1992.

[71] Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006*, volume 4502 of *LNCS*, pages 237–252. Springer, 2006.

[72] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The metacoq project. *Journal of Automated Reasoning*, pages 1–53, 2020.

[73] Michael Spivey and Jean-Raymond Abrial. *The Z notation*. Prentice Hall, 1992.

[74] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[75] Fairouz Tchier. *Sémantiques relationnelles démoniaques et vérification de boucles non déterministes*. PhD thesis, Université Laval, 1997.

[76] The Coq Development Team. The Coq Proof Assistant, version 8.11.0, January 2020.

[77] Alan Turing. On checking a large routine. In *Report of a Conference on 11i9h Speed Automatic Calculating Machines*, pages 67–69, 1949.

[78] Vladimir Voevodsky. The equivalence axiom and univalent models of type theory. *Talk at CMU*, pages 172–187, 2010.

[79] Maurice V. Wilkes. Memoirs of a computer pioneer. page 145. Massachusetts Institute of Technology, 1985.

[80] Niklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

Liste des Figures