Extraction de code fonctionnel certifié à partir de spécifications inductives

Pierre-Nicolas Tollitte

▶ **To cite this version:**

Pierre-Nicolas Tollitte. Extraction de code fonctionnel certifié à partir de spécifications inductives. Informatique et langage [cs.CL]. Conservatoire national des arts et metiers - CNAM, 2013. Français. NNT : 2013CNAM0895 . tel-00968607

HAL Id: tel-00968607

https://theses.hal.science/tel-00968607

Submitted on 1 Apr 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS | le cnam

École Doctorale Informatique, Télécommunications et Électronique

Centre d'étude et de recherche en informatique et communications (EA 4629)

THÈSE DE DOCTORAT

présentée par : **Pierre-Nicolas TOLLITTE**

soutenue le : **6 décembre 2013**

pour obtenir le grade de : **Docteur du Conservatoire National des Arts et Métiers**

Discipline / Spécialité : **Informatique**

Extraction de code fonctionnel certifié à partir de spécifications inductives

THÈSE DIRIGÉE PAR

Mme DUBOIS Catherine	*Professeur, ENSIIE (Directrice)*
M. DELAHAYE David	*Maître de conférences, CNAM (Codirecteur)*

RAPPORTEURS

M. LETOUZEY Pierre	*Maître de Conférences, Université Paris Diderot*
M. RUSU Vlad	*Chargé de Recherche, INRIA*

EXAMINATEURS

M. HERBELIN Hugo	*Directeur de Recherche, INRIA*
M. JAUME Mathieu	*Maître de Conférences, Université Pierre et Marie Curie*
M. LEROY Xavier	*Directeur de Recherche, INRIA*

À mes parents,
À Assma

Remerciements

Je remercie Pierre Letouzey et Vlad Rusu, qui ont accepté d'être rapporteurs de cette thèse, Hugo Herbelin, Mathieu Jaume qui ont fait partie du jury et Xavier Leroy qui l'a dirigé.

Je remercie mes directeurs de thèse Catherine Dubois et David Delahaye pour la qualité du travail qu'ils ont réalisé avec moi, les connaissances et les exigences qu'ils m'ont transmises, leur patience, la liberté qu'ils m'ont accordée ainsi que la confiance qu'ils m'ont témoignée.

Je remercie Tristan Crolard pour le partage de ses travaux.

Je remercie Benoît, Matthieu, Mélanie et William pour leur soutien et les bons moments partagés ensemble, au bureau et ailleurs.

Je remercie Zaynah Dargaye, qui m'a poussé à faire cette thèse.

Je remercie les professeurs et élèves de l'ENSIIE.

Je remercie mes parents et mes deux sœurs qui sont toujours d'un grand soutien.

Enfin, je remercie Assma pour ses conseils et sa motivation, prodigués avec justesse et générosité.

Résumé

Les outils d'aide à la preuve basés sur la théorie des types permettent à l'utilisateur d'adopter soit un style fonctionnel, soit un style relationnel (c'est-à-dire en utilisant des types inductifs). Chacun des deux styles a des avantages et des inconvénients. Le style relationnel peut être préféré parce qu'il permet à l'utilisateur de décrire seulement ce qui est vrai, de s'abstraire temporairement de la question de la terminaison, et de s'en tenir à une description utilisant des règles. Cependant, une spécification relationnelle n'est pas exécutable.

Nous proposons un cadre général pour transformer une spécification inductive en une spécification fonctionnelle, en extrayant à partir de la première une fonction et en produisant éventuellement la preuve de correction de la fonction extraite par rapport à sa spécification inductive. De plus, à partir de modes définis par l'utilisateur, qui permettent de considérer les arguments de la relation comme des entrées ou des sorties (de fonction), nous pouvons extraire plusieurs comportements calculatoires à partir d'un seul type inductif.

Nous fournissons également deux implantations de notre approche, l'une dans l'outil d'aide à la preuve Coq et l'autre dans l'environnement Focalize. Les deux sont actuellement distribuées avec leurs outils respectifs.

Mots clés : Spécifications exécutables, relations inductives, génération de code fonctionnel, génération de preuves, Coq, Focalize.

Abstract

Proof assistants based on type theory allow the user to adopt either a functional style, or a relational style (e.g., by using inductive types). Both styles have advantages and drawbacks. Relational style may be preferred because it allows the user to describe only what is true, discard momentarily the termination question, and stick to a rule-based description. However, a relational specification is usually not executable.

We propose a general framework to turn an inductive specification into a functional one, by extracting a function from the former and eventually produce the proof of soundness of the extracted function w.r.t. its inductive specification. In addition, using user-defined modes which label inputs and outputs, we are able to extract several computational contents from a single inductive type.

We also provide two implementations of our approach, one in the Coq proof assistant and the other in the Focalize environnement. Both are currently distributed with the respective tools.

Keywords : Executable specifications, inductive relations, functional code generation, proof generation, Coq, Focalize.

Table des matières

1 Introduction **19**

2 Notions préliminaires **25**

2.1 Induction 25

2.1.1 Raisonnement par récurrence 25

2.1.2 Récurrence bien fondée 26

2.1.3 Types de données inductifs 26

2.1.4 Relations inductives 27

2.1.5 Induction structurelle 28

2.2 Sémantique des langages de programmation 29

3 État de l'art **33**

3.1 Outils d'aide à la preuve avec une approche similaire à la nôtre 33

3.2 Formalismes et outils dédiés à la sémantique des langages 37

3.3 Positionnement 40

4 Approche informelle **43**

4.1 Approches relationnelle et fonctionnelle 43

4.1.1 Relations inductives 43

4.1.2 Style fonctionnel 44

4.1.3 Avantages et inconvénients 47

4.2 Passer d'une approche à l'autre 48
4.2.1 De la relation inductive à la fonction 48
4.2.2 De la fonction à la relation inductive 51

5 De la relation inductive au programme fonctionnel 53

5.1 Formalisation des relations inductives 53
5.1.1 Types de données inductifs 54
5.1.2 Relations inductives 54
5.2 Définitions et notations 55
5.3 Schéma d'extraction de base 57
5.3.1 Analyse de cohérence de mode 58
5.3.2 Génération de code MiniML 59
5.3.3 Principale limitation 60
5.4 Représentation de données pour l'extraction 60
5.4.1 Exemple d'extraction utilisant un Reltree 62
5.4.2 Liens entre Reltree, relation inductive et code généré 65
5.5 Schémas de compilation pour les Reltree 66
5.6 Propriétés et invariants des Reltree 67
5.6.1 Définition des Reltree 68
5.6.2 Propriétés de bonne formation des Reltree 68
5.7 Construction des arbres 71
5.7.1 Rappels et notations 71
5.7.2 Principe de l'algorithme 72
5.7.3 Décomposition de l'algorithme 73
5.7.4 Algorithme de construction d'un ensemble de Reltree 75
5.7.5 Propriétés de l'algorithme de construction des Reltree 80

5.8 Génération de code fonctionnel à partir des Reltree 86
5.8.1 Définition du langage intermédiaire MFL 86
5.8.2 Algorithme de génération de code 87
5.8.3 Propriétés du code obtenu 88
5.9 Extensions 89
5.9.1 Fonctions utilisées dans la définition des relations inductives 90
5.9.2 Connecteurs logiques 91
5.9.3 Utilisation de l'égalité 92
5.9.4 Non déterminisme 93
5.10 Relations mutuellement inductives 96

6 Extraction vers ML **99**
6.1 Plugin d'extraction native de Coq 99
6.1.1 Fonctionnement de l'extraction 100
6.1.2 Langage intermédiaire 100
6.2 Implantation 101
6.2.1 De MFL à MiniML 102
6.2.2 Limites 102

7 Extraction vers Coq **105**
7.1 Calcul des constructions inductives de Coq 105
7.2 Compilation vers Coq 106
7.2.1 Compilation des filtrages 107
7.3 Problèmes de terminaison 111
7.3.1 Récursion structurelle 111
7.3.2 Compteur d'itérations 111
7.4 Fonctions partielles et dépendances 112

7.4.1 Extraction en mode partiel 112
7.4.2 Extraction en mode complet 113
7.4.3 Dépendances 113
7.5 Implantation et résultats 114
7.6 Génération automatique des preuves 115
7.6.1 Tactiques de preuve de Coq 115
7.6.2 Schémas d'induction pour les fonctions 118
7.6.3 Principe général de la génération des preuves 118
7.6.4 Preuves de correction 119
7.6.5 Preuves de complétude 125

8 Extraction dans l'environnement Focalize 127

8.1 Présentation de Focalize 127
8.1.1 L'environnement Focalize 127
8.1.2 Spécification et implantation dans Focalize 128
8.1.3 Preuves dans Focalize 130
8.2 Relations dans Focalize 131
8.3 Implantation de l'extraction dans Focalize 132
8.4 Preuves de correction 134
8.5 Discussion sur les liens entre extraction et héritage 135

9 Résultats et performances 137

9.1 Résultats 137
9.2 Comparaison avec d'autres outils 139
9.2.1 Comparaison qualitative 140
9.2.2 Performances 142

Conclusion **153**

Bibliographie **157**

Annexes **171**

A Sémantique du langage MFL **171**

B Sémantique du sixième langage intermédiaire de MLCompCert **173**

C Spécification Ott du langage IMP **175**

Liste des tableaux

2.1 Commandes du langage IMP 27

2.2 Règles d'inférence pour les commandes du langage IMP 31

5.1 Étapes de l'algorithme $\mathcal{R}$ 74

5.2 Contrats des algorithmes de construction des Reltree 81

7.1 Sous-ensemble du langage CIC 106

7.2 Influence des dépendances sur les modificateurs de fonctions extraites 114

9.1 Fonctionnalités implantées 138

9.2 Résultats pour trois extractions vers OCaml sur des outils provenant d'autres projets 139

9.3 Fonctionnalités d'extraction 140

9.4 Classes de relations inductives traitées 141

9.5 Fonctionnalités d'exécution 141

9.6 Fonctionnalités supplémentaires 142

9.7 Syntaxe du langage IMP 143

9.8 Relations définissant l'exécution du langage IMP 144

9.9 Spécification de la relation *exec* du langage IMP écrite en Ott 145

9.10 Vitesses d'exécution comparées 147

Table des figures

2.1 Type command en Coq 27

2.2 Spécification de l'addition en Coq 28

2.3 Arbre de dérivation pour l'exécution d'une commande 31

4.1 Spécification de l'exécution des commandes du langage IMP en Coq 44

4.2 Interprète pour les commandes du langage IMP 46

4.3 Utilisation de la fonction exec_imp en OCaml 46

4.4 Fonction d'addition add12, et de soustraction add23 49

4.5 Preuve de correction pour la fonction add12 50

4.6 Preuve de complétude pour la fonction add12 50

4.7 Relation inductive générée à partir de la fonction add12 51

5.1 Sémantique de `While` 61

5.2 Sémantique de `Repeat` 62

5.3 Recherche d'un chemin dans un arbre binaire de recherche 63

5.4 Extraction de la relation inductive search avec le mode $\{1, 2\}$ vers OCaml . 64

5.5 Reltree de la relation inductive `search` en mode $\{1, 2\}$ 64

5.6 Schéma de compilation général 66

5.7 Schéma de compilation vers OCaml et Haskell 66

5.8 Ensemble de Reltree obtenu après la premère étape de construction (S_1) . . 79

5.9 Reltree résultant de l'insertion de $exec\_while1$ dans rt_1 80

5.10 Reltree pour la relation inductive `exec` 81

5.11 Spécification de la relation inductive fibo 90

5.12 Spécification équivalente de la relation inductive fibo 91

5.13 Reltree et fonction générés à partir de la relation inductive fibo 92

5.14 Une autre version de la recherche dans un arbre binaire 94

5.15 Code extrait à partir de la relation inductive de la figure 5.14 avec le mode $\{1, 2\}$ 96

5.16 Spécification des relations pair et impair 97

5.17 Fonctions extraites avec le mode $\{1\}$: pair et impair 97

7.1 Schéma de compilation vers Coq 107

7.2 Résultat de la compilation des filtrages de la fonction search12 109

7.3 Schéma de compilation des preuves 118

7.4 Fonction add12 générée automatiquement 123

7.5 Script de preuve généré automatiquement pour la fonction add12 124

8.1 Spécification de l'addition dans une espèce Focalize 131

8.2 Modification d'une spécification en utilisant l'héritage 132

8.3 Extraction de la fonction add12 dans une espèce Focalize 134

8.4 Extraction de la fonction add dans une espèce Focalize 134

9.1 Programme de test IMP 144

9.2 Performance comparée de Coq, Isabelle/HOL et Twelf 148

Chapitre 1

Introduction

Contexte

Les outils d'aide à la preuve sont des outils issus d'une volonté de formalisation des mathématiques, de la logique, et d'autres formalismes qui proviennent de l'informatique, comme les langages de programmation. Cette démarche de formalisation n'est pas nouvelle. Elle a commencé avec "Principia Mathematica" [Whitehead and Russell 1957], une œuvre en trois volumes d'Alfred North Whitehead et Bertrand Russell dont l'objectif est de déduire les théorèmes mathématiques connus à partir d'un ensemble d'axiomes bien définis et de règles de déduction précises. En cela, cet ouvrage participe à la naissance de la logique moderne. Néanmoins, si Principia Mathematica montre qu'il est théoriquement possible de formaliser les mathématiques, il est impossible de fournir l'intégralité des preuves de manière effective, à cause de la taille de certaines preuves.

Une réponse à ce problème a été donnée par l'informatique qui, en apportant une puissance de calcul inédite, donne l'espoir que la formalisation complète des mathématiques peut devenir possible. Si certains outils permettent de rechercher automatiquement des preuves, ils ne permettent pas de trouver "au hasard" les preuves des théorèmes les plus complexes car la puissance de calcul des ordinateurs est limitée. L'intelligence humaine reste donc nécessaire à la preuve de ces théorèmes.

Mélanger la puissance de calcul des ordinateurs et l'intelligence humaine dans la réalisation des preuves est l'objectif des outils d'aide à la preuve dont le plus ancien est Automath [de Bruijn 1970], initialement développé par Nicolaas Govert De Bruijn en

1967 ([Geuvers 2009]). Un objectif du projet était de développer un langage dans lequel il est possible d'exprimer toutes les mathématiques, et qui assure que tout ce qui est correctement écrit dans ce langage est correct pour les mathématiques. Pour cela, Automath place les preuves comme des objets de première classe du langage et les formules à prouver comme des types. Cette idée est connue dans le monde logique sous le nom d'isomorphisme de Curry-Howard, écrit en 1969 par Howard [Howard 1980], que l'on appelle également correspondance formule/type.

Automath était simplement un vérificateur de preuves, ces dernières devant être écrites entièrement. Les outils d'aide à la preuve qui ont suivi mettent des tactiques à disposition de l'utilisateur qui lui permettent de simplifier l'écriture des preuves en réalisant plusieurs pas de preuve en une seule commande.

De nombreux outils ont vu le jour depuis Automath,dont les outils actuels Coq [The Coq Development Team 2012], Isabelle/HOL [Nipkow et al. 2002] et Mizar sont de célèbres exemples. Ces derniers ont permis de réaliser des développements conséquents comme CompCert [Leroy 2009; Camilleri 1997; Gonthier et al. 2013], ...

Aujourd'hui, les outils d'aide à la preuve n'ont pas encore relevé tous les défis. Certains théorèmes, comme la transcendance de π, n'ont encore été prouvés avec aucun outil d'aide à la preuve [Wiedijk 2013]. En revanche, on sait que la formalisation des preuves est utile car la vérification d'une preuve écrite à la main n'est pas suffisante pour établir la validité d'un théorème complexe. La preuve de la conjecture de Kepler réalisée par Thomas Hales a, par exemple, été publiée après 4 ans de vérification par douze référés ; cependant, ils ne purent pas garantir complètement la preuve [Hales 2005b] et Thomas Hales a lancé un projet afin de valider mécaniquement cette preuve en entier [Hales 2005a].

L'outil d'aide à la preuve Coq est un outil développé par Inria depuis presque 25 ans. Il est basé sur le calcul des constructions inductives qui permet en particulier de décrire des relations de manière inductive à travers un ensemble de propriétés. Il permet également de définir des fonctions exécutables. En revanche, les relations définies de manière inductive ne peuvent pas être exécutées, ce qui confronte l'utilisateur au choix entre relations non exécutables, sur lesquelles il est parfois plus facile de raisonner, et fonctions exécutables. Une solution possible consiste à écrire dans un développement Coq la spécification de ma-

nière relationnelle et de manière fonctionnelle et à montrer l'équivalence des deux versions. Cela permet de bénéficier des avantages des relations et des fonctions, mais représente un travail fastidieux.

Cette problématique existe également dans d'autres outils d'aide à la preuve comme par exemple Isabelle/HOL pour lequel un outil d'extraction de fonctions à partir de relations définies de manière inductive existe déjà. Néanmoins, ce dernier est spécifique au formalisme d'Isabelle/HOL et ne peut pas facilement être adapté à Coq. Notre objectif est de développer un cadre générique utilisable par les outils de preuve pour l'extraction de fonctions et de preuves à partir de relations définies de manière inductive et de formaliser ce processus d'extraction.

Motivations

Le mécanisme d'extraction que nous proposons peut avoir deux types d'utilisation. Le premier consiste à extraire des fonctions dans le même formalisme que celui qui a servi à décrire les relations dont sont extraites les fonctions. L'extraction automatique offre alors un gain direct : elle permet d'exécuter la relation sans avoir à écrire la fonction manuellement. Ce besoin apparaît clairement dans plusieurs développements Coq conséquents, comme dans [Blazy and Leroy 2009] et [Campbell 2012]. Toujours avec le même objectif, nous essayons également de produire automatiquement la preuve de correction et de complétude des fonctions extraites par rapport à leur spécification. La génération automatique du code des fonctions et des preuves rendent les développements Coq qui l'utilisent beaucoup plus faciles à maintenir. En effet, le code peut être généré de nouveau rapidement dès qu'une modification est apportée à la relation. Enfin, une relation définie de manière inductive peut contenir plusieurs comportements calculatoires, et il est alors possible d'extraire plusieurs fonctions à partir d'une seule relation.

Le second type d'utilisation concerne les fonctions extraites dans un formalisme différent des relations qui leur servent de spécification. Par exemple, nous offrons la possibilité d'extraire du code OCaml à partir d'un développement écrit en Coq. Cela permet un prototypage et l'animation rapide d'une spécification afin de la vérifier, ou de l'exécuter, et ainsi

de se rendre compte plus tôt dans le cycle de développement d'erreurs éventuelles. Le fait d'utiliser un formalisme externe peut aussi être utile afin de bénéficier de fonctionnalités du second formalisme qui ne sont pas présentes dans le premier. Par exemple, OCaml permet d'exécuter des fonctions qui ne terminent pas nécessairement, ce qui n'est pas le cas de Coq. On peut ainsi extraire un interprète en OCaml à partir de la sémantique d'un langage impératif qui comporte des boucles écrite en Coq, alors que les programmes de ce langage impératif ne terminent pas toujours.

Contributions

La première contribution de cette thèse est la formalisation d'un certain nombre d'algorithmes et de représentations intermédiaires associées. Le processus d'extraction qui y est décrit possède des interfaces clairement définies afin qu'il puisse être utilisé dans différents outils de preuve.

J'ai réalisé une première implantation de cette approche dans un plugin pour Coq. Ce dernier est disponible dans les contributions de Coq depuis la version 8.4 [Tollitte et al. 2013a]. Il définit deux nouvelles commandes, dont l'une permet l'extraction vers le langage OCaml et l'autre vers Coq. Lorsque l'extraction est réalisée vers OCaml, un fichier séparé est produit, qui contient le résultat de toutes les commandes d'extraction réalisées. L'utilisateur peut demander à ce que les dépendances des fonctions extraites le soient également, afin que soit produit un fichier OCaml se suffisant à lui-même. L'extraction vers Coq permet de fournir une implantation qui est directement utilisable dans le même fichier source que la spécification. La fonction qui est définie, même si elle n'est pas écrite dans le fichier, est utilisable comme n'importe quelle autre fonction. Il est possible de l'exécuter, de prouver des propriétés la concernant, de l'afficher, ... Pour l'instant, seules les fonctions récursives structurelles sont supportées. Nous fournissons cependant une astuce afin de supporter l'extraction des autres fonctions, en les rendant structurellement récursives. Pour cela, nous modifions leur type de retour et ajoutons un compteur (premier argument de la fonction) qui décroît à chaque appel de la fonction. Dans certains cas, nous générons également des lemmes de correction de la fonction extraite par rapport à sa spécification. Ces lemmes, comme les fonctions, peuvent être manipulés et affichés par l'utilisateur, qui peut ainsi avoir

parfaitement confiance dans le code extrait. Bien qu'il soit récent, ce plugin est utilisable dans le cadre de développements non triviaux. Un catalogue d'exemples est fourni avec le plugin, et nous illustrons chacun des chapitres de ce manuscrit avec ces derniers.

J'ai également développé une seconde implantation au sein de l'environnement Focalize [http ://focalize.inria.fr/ 2013], validant ainsi la possibilité de généraliser l'approche à d'autres outils. L'environnement Focalize propose un langage de spécification avec des traits qui proviennent du paradigme objet. Il permet à ce titre d'utiliser l'héritage et la redéfinition de fonctions. L'extraction permet de fournir une implantation éventuellement temporaire, dont on peut donner une version plus optimisée ou plus spécifique, plus tard dans le développement. Cependant, Focalize ne possède pas nativement de construction permettant de définir des relations inductives. Les déclarations de fonctions et la possibilité de caractériser ces déclarations avec des propriétés écrites dans la logique du premier ordre suffisent néanmoins à simuler la notion de relation inductive manquante et offrent une alternative pour l'extraction.

Les outils proposés présentent des limitations quant à la forme des spécifications. Ces limitations sont décrites dans la formalisation.

Une publication internationale [Tollitte et al. 2012] décrit entièrement la formalisation dans le cas spécifique de la génération de code Coq, ainsi que des preuves de correction de ces fonctions. Une autre publication [Delahaye et al. 2010] décrit l'implantation dans Focalize, mais utilise un algorithme de génération de code plus ancien.

Plan du manuscrit

Le chapitre 2 présente les notions utiles à la compréhension des chapitres suivants. Ces dernières concernent essentiellement l'induction et la sémantique des langages.

Le chapitre 3 détaille l'état de l'art des outils d'extraction pour les outils d'aide à la preuve. Il aborde également une famille d'outils similaires, dédiés à la sémantique des langages de programmation, principale source d'exemples pour notre outil.

Le chapitre 4 résume notre approche de manière informelle en rappelant les objectifs et enjeux de celle-ci. Il permet au lecteur de comprendre le processus d'extraction dont

l'automatisation fait l'objet des chapitres suivants, en déroulant sur un exemple complet l'écriture manuelle d'une fonction et des preuves de correction pour cette fonction.

Le chapitre 5 constitue le chapitre central de ce manuscrit. Il commence par résumer une première approche, travaux auxquels cette thèse fait suite. Il formalise ensuite le passage d'une relation définie de manière inductive à une fonction. Les chapitres 6, 7 et 8 décrivent les traductions réalisées depuis un langage fonctionnel commun, cible de notre processus d'extraction, vers les langages cibles : OCaml, Coq et Focalize. Le chapitre 7 traite également de la génération des preuves de correction de la fonction extraite par rapport à sa spécification dans le cas de la production de fonctions Coq.

Le chapitre 9 décrit un exemple complet, montrant les possibilités offertes par notre outil, puis il réalise une étude comparative avec deux autres outils : Isabelle/HOL et Twelf.

Enfin, la conclusion dresse un bilan du travail effectué et présente les perspectives d'évolution de l'approche présentée dans cette thèse.

Chapitre 2

Notions préliminaires

Ce chapitre contient des rappels et précise quelques définitions qui seront utiles pour comprendre les chapitres suivants. Nous commençons par présenter le raisonnement par induction et les notions qui s'y rapportent (types de données et relations inductives) dans la section 2.1. Ensuite, nous faisons un rappel sur la sémantique des langages de programmation qui se formalise généralement en utilisant des relations inductives et de laquelle nous tirons la plus part de nos exemples, dans la section 2.2.

2.1 Induction

Les preuves de propriétés sur les programmes utilisent souvent une technique de preuve appelée induction ou récurrence. Il s'agit plus en réalité d'une famille de techniques de preuves dont les plus courantes sont : le raisonnement par récurrence et l'induction structurelle qui sont des cas particuliers de l'induction bien fondée.

2.1.1 Raisonnement par récurrence

Le raisonnement par récurrence est une forme de raisonnement bien connue en mathématiques. Elle vise à démontrer une propriété $P(n)$ pour tous les entiers naturels n en démontrant les deux énoncés suivants :

— $P(0)$

— $\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)$

Une fois ces deux points démontrés, on peut conclure que la propriété est vraie pour tous les entiers naturels : $\forall n \in \mathbb{N}, P(n)$.

Une variation de ce raisonnement est la récurrence forte, dont on peut montrer l'équivalence avec la définition précédente. Elle est définie comme suit :
$\forall n \in \mathbb{N}, P(n) \Leftrightarrow \forall n \in \mathbb{N}, (\forall k \in \mathbb{N}, k < n \Rightarrow P(k)) \Rightarrow P(n)$

On peut également appliquer ces deux principes à partir d'un certain rang n_0.

2.1.2 Récurrence bien fondée

La récurrence bien fondée est une généralisation du principe de récurrence. Elle reprend l'énoncé de la récurrence forte et permet de s'abstraire des entiers naturels et d'étendre le raisonnement à tous les ensembles bien fondés ou les ensembles munis d'une relation bien fondée[1]. Si $\prec$ est une relation bien fondée sur l'ensemble A, alors on a :
$\forall a \in A, P(a) \Leftrightarrow \forall a \in A, (\forall b \in A, b \prec a, P(b)) \Rightarrow P(a)$.

2.1.3 Types de données inductifs

Mathématiquement, un ensemble défini inductivement est le plus petit ensemble vérifiant un certain nombre de propriétés ou règles. Certaines de ces règles affirment que des éléments donnés appartiennent à l'ensemble, et d'autres permettent de construire de nouveaux éléments à partir d'éléments déjà construits.

On peut ainsi définir l'ensemble des entiers naturels $\mathbb{N}$ avec les règles suivantes :

— $0 \in \mathbb{N}$

— Si $n \in \mathbb{N}$ alors $succ(n) \in \mathbb{N}$ où $succ$ est la fonction qui à n associe $n + 1$.

Les types inductifs [Coquand and Paulin 1988; Pfenning and Paulin-Mohring 1989] sont des types de données définis par la donnée d'un ensemble de constructeurs, qui correspondent aux règles mentionnées ci-dessus. Nous entendons par type de données inductif, un type qui admet pour valeurs celles qui appartiennent au plus petit ensemble que l'on peut construire à partir d'un ensemble d'éléments de base, et en appliquant un nombre fini de fois les règles de construction. Les éléments de base sont généralement des constructeurs

1. Une relation $\prec$ est bien fondée sur A s'il n'existe pas de suite infinie (a_n) d'éléments de A telle qu'on ait $a_{n+1} \prec a_n$ pour tout n

d'arité zéro, et les règles de construction sont déduites des constructeurs d'arité non nulle.

Par exemple, pour les commandes du langage impératif IMP décrites dans le tableau 2.1, on peut définir en Coq le type inductif de la figure 2.1. Les constructeurs sont donnés dans le même ordre. `Skip` est un constructeur d'arité zéro, et `Sequ`, `Test` et `While` sont trois constructeurs d'arité non nulle.

$$c ::= \textbf{skip} \mid X := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c$$

où X est une variable, b une expression booléenne et b une expression arithmétique.

TABLE 2.1 – Commandes du langage IMP

```
Inductive command : Set :=
  | Skip   : command
  | Affect : variable → arith_expr → command
  | Sequ   : command → command → command
  | Test   : bool_expr → command → command → command
  | While  : bool_expr → command → command.
```

où `variable` est le type des variables, `bool_expr` celui des expressions booléennes et `arith_expr` celui des expressions arithmétiques.

FIGURE 2.1 – Type command en Coq

Les types de données inductifs peuvent être utilisés pour construire des filtrages. Un filtrage par motif permet de vérifier si un terme possède une structure donnée, ainsi que la présence de sous-structures spécifiques et d'identifier leur valeur. La valeur que possède le filtrage **match** `t` **with** `| p1 → e1 | ... | pn → en` est la valeur de `ei` si `t` a la structure du motif de filtrage `pi` et si pour tout `j` $<$ `i`, `t` n'a pas la structure du motif de filtrage `pj`. On peut de plus utiliser toutes les variables liées par `pi` pour calculer `ei`.

2.1.4 Relations inductives

La notion de relation définie de manière inductive est proche de celle des types de données inductifs. Au lieu de définir un ensemble de valeurs, on utilise l'induction pour définir un prédicat à partir d'un ensemble de règles. On définit une relation inductive en donnant son type et un ensemble de propriétés qui la caractérisent. La relation inductive est alors définie comme étant le plus petit prédicat vérifiant ces règles. Les règles sont appelées constructeurs de la relation inductive. Les relations inductives sont définies en

Coq en utilisant le même mot-clé que les types de données inductifs : **`Inductive`**.

Les définitions inductives présentées dans ce chapitre et le suivant sont écrites en Coq. La définition que nous retenons pour notre formalisation dans le chapitre 5 est un sous-ensemble des définitions inductives que l'on peut écrire en Coq.

La relation de la figure 2.2, par exemple, est une relation entre trois entiers naturels et spécifie que le troisième argument est la somme des deux premiers. Il s'agit d'une relation à trois arguments de type `nat`, qui possède deux constructeurs `O` et `S`. La relation inductive est définie comme le plus petit prédicat vérifiant les règles `addO` et `addS`, et seuls les cas où $n_1 + n_2 = n_3$ sont décrits dans sa définition. Le constructeur `addO` précise que $n + 0 = 0$ et `addS` que si $n + m = p$ alors $n + (m + 1) = p + 1$.

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

FIGURE 2.2 – Spécification de l'addition en Coq

2.1.5 Induction structurelle

Le principe d'induction est le pendant logique du raisonnement par récurrence des mathématiques. L'induction structurelle est liée à la définition du type inductif. Pour vérifier une propriété par induction structurelle, on vérifie la propriété sur les constructeurs d'arité zéro, puis on suppose cette propriété vraie pour certains éléments et on vérifie qu'elle reste vraie lorsque l'on applique un constructeur d'arité non nulle à ces éléments. Cela est valable pour les types de données inductifs comme pour les relations inductives.

Pour les commandes du langage IMP, le principe d'induction est le suivant : Pour prouver une propriété P pour toute commande c, on doit montrer :

— $P(\textbf{skip})$

— $\forall X, \forall a, P(\mathbf{X} := \mathbf{a})$

— $\forall c_0\ c_1, P(c_0) \wedge P(c_1) \Rightarrow P(c_0; c_1)$

— $\forall b, \forall c_0\ c_1, P(c_0) \wedge P(c_1) \Rightarrow P(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1)$

— $\forall b, \forall c, P(c) \Rightarrow P(\textbf{while } b \textbf{ do } c)$

De même, pour la relation inductive `add` : Pour prouver la propriété $\forall a\ b\ c, add\ a\ b\ c \rightarrow P\ a\ b\ c$, on doit montrer :

— $\forall a, P\ a\ 0\ a$

— $\forall a\ b\ c, add\ a\ b\ c \rightarrow P\ a\ b\ c \rightarrow P\ a\ (S\ b)\ (S\ c)$

2.2 Sémantique des langages de programmation

La sémantique formelle d'un langage permet de définir mathématiquement la signification des constructions présentes dans le langage (expressions, instructions, ...). Le champ d'application de la sémantique des langages de programmation n'est pas limité aux compilateurs. Elle constitue un prérequis pour caractériser et prouver des propriétés sur les programmes. Le domaine qui couvre la recherche de propriétés sur les programmes s'appelle l'analyse statique de programmes dans lequel on retrouve les techniques de model checking [Clarke et al. 2009], d'analyse de flots de données [Kildall 1973], d'interprétation abstraite [Cousot and Cousot 1977] et la preuve. Parmi les outils qui permettent d'écrire des sémantiques et des interprètes, on trouve les outils d'aide à la preuve ainsi que des outils dédiés tels que XText [Eclipse.org 2013]. L'aboutissement des travaux en preuve formelle est sans doute l'élaboration de compilateurs vérifiés formellement dont l'exemple le plus récent et conséquent est CompCert [Leroy 2009].

Il existe principalement trois approches de la sémantique formelle :

— la sémantique opérationnelle,

— la sémantique dénotationnelle,

— et la sémantique axiomatique.

La sémantique opérationnelle décrit le comportement d'un programme, en spécifiant comment il s'exécute sur une machine abstraite. C'est cette approche qui nous intéresse particulièrement ici, parce que nous pouvons générer des fonctions qui réalisent de manière effective ce que les spécifications décrivent de manière abstraite. La sémantique dénotationnelle associe à un programme un objet mathématique (une fonction par exemple) qui dénote son sens. Pour la sémantique axiomatique, le sens d'un programme est donné par ce qu'il est possible de prouver à partir des axiomes qui définissent la sémantique. Nous n'étudierons pas les sémantiques dénotationnelle et axiomatique.

Gordon Plotkin est le premier à avoir montré que la sémantique SOS ("structural operational semantics"), ou sémantique à petits pas, est utilisable en pratique pour définir la sémantique de langages complets [Plotkin 1981]. Le terme SOS vient du fait que l'écriture des règles qui définissent la sémantique est dirigée par la syntaxe (structure) du langage.

Un groupe de chercheurs sous la direction de Gilles Kahn, s'est concentré sur l'évaluation ou l'exécution vers une valeur ou un état final, ce qu'ils appellent la sémantique naturelle [Clément et al. 1986; Kahn 1987]. Sémantique naturelle, parce que les arbres de dérivation sont proches des preuves faites en déduction naturelle [Prawitz 1965]. La sémantique naturelle, aussi appelée sémantique à grands pas, est un formalisme qui a été couramment utilisé au cours des 25 dernières années pour décrire les langages de programmation.

Prenons un exemple extrait de [Winskel 1993] pour illustrer la sémantique à grands pas. Nous ne traduisons ici qu'une partie de l'exemple qui est donné dans ce livre. Le tableau 2.2 donne la sémantique de la relation d'exécution des commandes du langage IMP. Le langage IMP est un langage impératif élémentaire, dans lequel on trouve des variables X, des expressions booléennes b et arithmétiques a, et des commandes c. Les notations X, b, a et c sont des métavariables : nous utiliserons par exemple a, a_0, a_1, ... pour désigner des expressions arithmétiques. On définit la notion d'état pour un programme IMP. Un état σ est une fonction qui va de l'ensemble des variables vers l'ensemble des valeurs entières. On suppose l'existence de deux relations d'évaluation, l'une pour les expressions arithmétiques ($\langle a, \sigma\rangle \to n$ où n est un entier) et l'autre pour les expressions booléennes ($\langle b, \sigma\rangle \to t$ où t est **true** ou **false**). Ces deux relations peuvent être définies à l'aide des règles d'inférence, mais nous nous contentons ici de la définition de la relation d'exécution des commandes ($\langle c, \sigma\rangle \to \sigma$) décrite dans le tableau 2.2. Une règle d'inférence comporte deux parties, séparées par un trait horizontal. La partie supérieure est nommée prémisse(s) et la partie inférieure conclusion. Une règle sans prémisse, comme (skip), est appelée axiome. La règle (séquence) peut être lue de la manière suivante : si l'exécution de c_0 dans l'état σ produit l'état σ'' et si l'exécution de c_1 dans l'état σ'' produit l'état σ' alors l'exécution de la séquence de commande $c_0; c_1$ dans l'état σ produit l'état σ'.

Pour simuler l'exécution d'une commande à partir de ces règles, on construit un arbre

$$\text{(skip)}\ \frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}$$

$$\text{(affectation)}\ \frac{\langle a, \sigma\rangle \to m}{\langle X := a, \sigma\rangle \to \sigma[m/X]}$$

$$\text{(séquence)}\ \frac{\langle c_0, \sigma\rangle \to \sigma'' \qquad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \sigma'}$$

$$\text{(test}_1\text{)}\ \frac{\langle b, \sigma\rangle \to \textbf{true} \qquad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}$$

$$\text{(test}_2\text{)}\ \frac{\langle b, \sigma\rangle \to \textbf{false} \qquad \langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}$$

$$\text{(boucle}_1\text{)}\ \frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}$$

$$\text{(boucle}_2\text{)}\ \frac{\langle b, \sigma\rangle \to \textbf{true} \qquad \langle c, \sigma\rangle \to \sigma'' \qquad \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}$$

TABLE 2.2 – Règles d'inférence pour les commandes du langage IMP

de dérivation dont les nœuds sont les règles d'inférence et les feuilles des axiomes. Pour ce faire, on part de la conclusion, puis on applique des règles sur les prémisses jusqu'à trouver des axiomes. La figure 2.3 montre un extrait du déroulement de l'exécution de la commande **if** $X_0 = 3$ **then** $X_1 := X_0 + 1$ **else skip**, dans un état σ, tel que $\sigma(X_0) = 3$ et $\sigma(X_1) = 0$. Nous ne précisons pas les évaluations des expressions dans cet arbre de dérivation car nous n'avons pas donner les règles correspondantes.

$$test_1\ \frac{\dfrac{\cdots}{\langle X_0 = 3, \sigma\rangle \to \textbf{true}} \quad \text{séquence} \quad \dfrac{\dfrac{\cdots}{\langle X_0 + 1, \sigma\rangle \to 4}}{\langle X_1 := X_0 + 1, \sigma\rangle \to \sigma[4/X_1]}}{\langle \textbf{if } X_0 = 3 \textbf{ then } X_1 := X_0 + 1 \textbf{ else skip}, \sigma\rangle \to \sigma[4/X_1]}$$

FIGURE 2.3 – Arbre de dérivation pour l'exécution d'une commande

La sémantique définie dans le tableau 2.2 est appelée sémantique à grands pas car elle spécifie la transition entre une configuration ⟨commande, état⟩ et un état final : celui qui résulte de l'exécution des commandes du programme.

La sémantique à petits pas consiste à décrire les étapes qui permettent de passer d'une configuration donnée à une configuration plus simple (un pas de réduction ayant été effec-

tué). Il faut ensuite itérer le processus afin d'obtenir le même résultat que la sémantique à grand pas, après un certain nombre d'étapes, ou pas de réduction.

Chapitre 3

État de l'art

L'objectif de l'état de l'art est tout d'abord de dresser un bilan des outils existants pour l'exécution et l'animation de spécifications. Il s'agit dans certains cas d'extraction de code exécutable et dans d'autres cas d'exécution directe de la spécification. Nous présentons ces outils dans la section 3.1, en étudiant de façon détaillée les outils d'aide à la preuve. La sémantique des langages de programmation constituant une application importante pour notre outil d'extraction, nous donnons ensuite un aperçu de certains outils spécifiques à la sémantique des langages dans la section 3.2. Ces derniers effectuent, d'une certaine manière, un travail similaire en fournissant un interprète et d'autres programmes exécutables à partir de la sémantique d'un langage. Enfin, nous donnons le positionnement de notre approche par rapport à ces différents outils dans la section 3.3.

3.1 Outils d'aide à la preuve avec une approche similaire à la nôtre

Nous nous intéressons ici aux outils de preuve qui permettent d'exécuter des relations inductives, ou plus généralement des spécifications contenant des comportements calculatoires. Nous distinguons trois types de fonctionnalités intéressantes :

1. l'exécution directe, qui ne nécessite pas de génération de code ;
2. la génération de code dans le formalisme de la spécification ;
3. et la génération de code exécutable dans un langage ou un formalisme différent de celui dans lequel est écrite de la spécification.

Le troisième type d'exécution ne concerne pas nécessairement uniquement les relations inductives et peut parfois être utilisé pour extraire un programme fonctionnel vers un formalisme différent, la plupart du temps pour extraire du code vers un langage plus efficace comme OCaml ou obtenir un code indépendant de l'outil de spécification.

HOL ("Higher Order Logic") est un formalisme logique utilisé par une grande famille d'outils d'aide à la preuve, dont fait partie HOL4 [Slind and Norrish 2008]. Ce sont des successeurs de LCF [Gordon et al. 1979] pour lequel a été introduit le langage ML destiné à écrire des tactiques de preuve. Les tactiques sont des petits programmes qui font progresser la preuve. Un type abstrait est défini pour les tactiques, ainsi que des opérations sur ce type abstrait qui correspondent aux règles d'inférence du système de déduction utilisé. Le système de types de ML assure que les théorèmes sont dérivés en utilisant seulement les règles d'inférence données par les opérations sur le type abstrait des tactiques. Le système Isabelle [Wenzel and Berghofer 2013] est un outil d'aide à la preuve générique qui peut être instancié avec un système de déduction comme par exemple HOL dans le cas d'Isabelle/HOL. Les extensions d'Isabelle/HOL lui confèrent de très nombreuses fonctionnalités, parmi lesquelles on retrouve la possibilité de définir des types de données inductifs et des relations inductives [Berghofer and Wenzel 1999]. Isabelle/HOL permet également d'exécuter des programmes fonctionnels écrits par l'utilisateur. Une première extension d'Isabelle/HOL permettait d'extraire du code fonctionnel à partir de types de données inductifs, de relations inductives et de définitions de fonctions [Berghofer and Nipkow 2000]. La nouvelle approche consiste à transformer les relations inductives en équations logiques [Berghofer et al. 2009] et à utiliser le nouveau générateur de code d'Isabelle/HOL [Haftmann and Nipkow 2010] pour produire du code fonctionnel en utilisant ces équations. Cet outil, s'il permet effectivement d'exécuter des spécifications et de raisonner sur les fonctions extraites, ne partage pas exactement les mêmes objectifs que nos travaux. Tout d'abord, il est spécifique à Isabelle/HOL et ne pourrait pas être implanté dans Coq, car il utilise le générateur de code d'Isabelle/HOL pour transformer des équations en code exécutable. Notre approche, au contraire, se veut généralisable à différents outils. Enfin, une relation inductive étant définie par un ensemble de règles, il se peut que la fonction extraite ne soit pas déterministe et que cette dernière renvoie plusieurs résultats différents suivant la règle

que l'on choisit d'appliquer. Notre outil, au contraire, vérifie le caractère déterministe des spécifications et en extrait une fonction qui ne renvoie qu'un seul résultat, alors que l'outil d'Isabelle/HOL ne vérifie pas l'unicité du résultat de la fonction extraite qui produit une liste contenant un nombre variable de valeurs. La preuve d'équivalence entre l'expression compilée et la définition inductive est résumée dans [Bulwahn 2009], et des arguments informels sont donnés quant à la complétude de la compilation des définitions inductives.

Coq est un outil d'aide à la preuve qui permet d'exprimer des spécifications logiques et inductives, des fonctions exécutables et des théorèmes au sein du même formalisme : le calcul des constructions inductives (CIC). Un plugin d'extraction [Paulin-Mohring 1989b; Letouzey 2004] offre la possibilité d'extraire les types de données inductifs et les fonctions. Il permet également de produire du code à partir des preuves mais il ignore les parties logiques dont font partie les relations inductives. La certification de ce processus d'extraction est abordée dans [Glondu 2012]. Même si la certification complète de l'extraction ne semble pas d'actualité, essentiellement à cause de l'absence d'une formalisation "précise et fidèle" du CIC, une vérification du code *a posteriori* y est proposée. Il s'agit d'une perspective intéressante pour envisager la génération de code efficace et entièrement certifiée à partir de Coq, le compilateur certifié de [Dargaye 2009] faisant le lien entre MiniML et le code machine. Un prototype d'extraction pour les relations inductives a été développé et prouvé [Delahaye et al. 2007]. Il permet de générer des fonctions qui réalisent les comportements calculatoires que l'on retrouve dans une classe spécifique de relations inductives. La correction et la complétude du mécanisme d'extraction font l'objet d'une preuve papier. L'implantation proposée était en revanche limitée par d'importantes restrictions imposées sur les spécifications dont elle pouvait extraire du code fonctionnel mais plusieurs optimisations possibles étaient indiquées afin de traiter davantage de spécifications. Cette thèse fait suite à ces travaux initiaux.

L'environnement de développement de programmes certifiés Focalize offre un langage fonctionnel avec des traits orientés objet. Il permet d'écrire des propriétés en utilisant la logique du premier ordre et de prouver ces dernières au sein d'un seul environnement cohérent. Un outil de preuve automatique, Zenon [Bonichon et al. 2007], est intégré à Focalize, auquel l'utilisateur fournit des squelettes de preuve. Focalize ne permet pas de

définir des relations inductives, mais il constitue pourtant la deuxième cible de nos travaux. L'implantation de notre approche dans cet environnement [Delahaye et al. 2010] montre qu'elle peut être généralisée à d'autres outils, même s'ils ne permettent pas de définir nativement des relations inductives.

Dans un registre différent, Maude est un langage basé sur la réécriture. En particulier, Maude peut être utilisé pour décrire des sémantiques opérationnelles en traduisant les règles d'inférence vers des règles de réécriture conditionnelle [Verdejo and Martí-Oliet 2006]. Cette démarche est appuyée par des études de cas très complètes [Verdejo and Martí-Oliet 2005] qui mettent en avant des techniques générales pour la description de sémantiques dans Maude.

Twelf [Frank Pfenning and Carsten Schuermann 2002] est un langage qui permet de spécifier, d'implanter et de prouver des propriétés sur des systèmes déductifs, tels que des langages de programmation ou des logiques. Il permet d'écrire ces spécifications de manière déclarative dans le style des prédicats Prolog. Il est basé sur le concept de "framework" logique, dont l'idée est de présenter un formalisme logique dans une théorie des types d'ordre supérieur. Il implante le "framework" logique LF [Harper et al. 1993], qui repose sur la théorie des types dépendants. Twelf ne possède pas de coupure, contrairement à Prolog (ou elle est dénotée par un point d'exclamation), ce qui le rend moins adapté pour faire de la programmation logique, mais il est très efficace pour formaliser des langages de programmation. Twelf offre la possibilité d'attribuer des modes aux spécifications permettant ainsi d'obtenir gratuitement un interprète directement exécutable dans le cas de la sémantique d'un langage. L'exécution n'est, en revanche, pas toujours efficace, comme mentionné dans [Appel et al. 2012]. En plus de l'exécution et de la possibilité de réaliser des preuves manuelles sur les spécifications, Twelf fournit des outils automatiques pour tester les propriétés de terminaison et de complétude, mais il ne permet pas d'extraire de code.

PVS [Owre et al. 1999] est un autre outil de preuve développé depuis de nombreuses années. Comme Isabelle/HOL, il est basé sur la logique d'ordre supérieur et dispose de nombreuses extensions. Son langage de spécification permet de définir des relations inductives mais elles ne sont pas exécutables.

Minlog [Berger et al. 2011] est un outil d'aide à la preuve basé sur la logique du premier ordre. Il est très avancé dans l'extraction de programmes à partir de preuves, notamment à partir de preuves basées sur des définitions inductives et co-inductives. Ces définitions peuvent être des relations inductives. En revanche, Minlog ne permet pas d'extraire des programmes directement à partir des définitions inductives, comme le permet Isabelle/HOL ou l'exécution de relations via des modes comme dans Twelf.

Mercury [Somogyi et al. 1994], comme Twelf, est un langage qui combine la programmation déclarative et la programmation logique. Il permet de définir des modes pour chaque prédicat, et de préciser des propriétés pour ces modes, parmi lesquelles le déterminisme et la complétude. Ces propriétés sont automatiquement vérifiées pas le compilateur et ont un intérêt double. Elles fournissent de l'aide aux programmeurs qui peuvent s'assurer du comportement de leur programme, et autorisent aussi le compilateur à pratiquer certaines optimisations.

3.2 Formalismes et outils dédiés à la sémantique des langages

De nombreux outils existent pour décrire les sémantiques, prouver des propriétés sur ces dernières, ou bien fournir des interprètes qui leur correspondent. Cette section a pour but de donner un aperçu de quelques-uns de ces outils. Nous nous intéresserons plus particulièrement à ceux qui ont pour objectif de fournir automatiquement des interprètes à partir de spécifications étant donné que c'est la motivation première des travaux de cette thèse.

Mentor [Donzeau-Gouge et al. 1980] est l'un des premiers outils de développement de programmes (Pascal notamment) dont la spécificité consiste à manipuler les arbres de syntaxe abstraite du langage. Centaur [Borras et al. 1988] est un générateur d'environnements interactifs générique qui étend l'approche de Mentor. À partir de la spécification formelle d'un langage de programmation, il peut générer les outils que l'on retrouve dans un environnement de développement intégré : éditeur, interprète, outil pour déboguer. Les spécifications des langages sont exprimées dans deux langages, Metal et Typol, qui décrivent respectivement la syntaxe et la sémantique. L'exécution de la spécification est réalisée en

utilisant un moteur Prolog. Le système Centaur lui-même est programmé en Lisp. L'utilisateur de Centaur peut utiliser l'environnement produit et son interface graphique, ou en accédant directement aux fonctions utilisées par cette interface afin de produire ses propres outils. Il n'est plus raisonnablement utilisable aujourd'hui car il n'est plus maintenu.

Une deuxième implantation a été réalisée en intégrant à Centaur les grammaires attribuées [Knuth 1968] pour former le système Minotaur [Attali and Parigot 1994].

D'autres approches similaires existent telles que le méta-environnement ASF+SDF [Klint 1993]. Il fait le lien entre ASF qui exploite la sémantique et la syntaxe abstraite et SDF qui lie la syntaxe concrète à la syntaxe abstraite d'un langage, permettant ainsi de mettre en corrélation le code source d'un programme et sa sémantique.

Une alternative à Typol, RML [Pettersson 1994], emprunte à Prolog les flots de contrôle, les variables logiques et l'unification auquel il joint les types polymorphes, les types de données et le filtrage de ML. Un compilateur ayant pour cible le langage C a été réalisé pour RML et permet d'obtenir de meilleures performances qu'avec un interprète Prolog classique, en effectuant des optimisations sur une représentation intermédiaire [Pettersson 1996]. La dernière version de RML a été diffusée en 2007, et le dernier manuel en 2006 [Peter Frizson 2006]. Un compilateur pour le langage Modelica [Fritzson and Engelson 1998] a été réalisé avec RML [Kågedal and Fritzson 1998]. Le logiciel OpenModelica incorpore une version du compilateur Modelica écrite en RML.

Une approche différente, genèse des travaux présentés dans [Delahaye et al. 2007] et ici, consiste à extraire des programmes fonctionnels à partir des spécifications en sémantique naturelle. Un premier pas a été fait avec un compilateur vers ML [Dubois and Gayraud 1999]. Chaque relation définie en sémantique naturelle possède un certain nombre d'arguments. Dans les règles d'inférence qui définissent ces relations, des variables peuvent apparaître dans les termes qui constituent ces arguments. Quand on compile une règle vers un programme Prolog, on demande à Prolog de calculer les valeurs des variables des termes libres à partir des termes que l'on a déjà instanciés. Cette nouvelle approche de compilation vers un langage fonctionnel définit la notion de "direction" qui oriente les relations. Une direction est un ensemble d'indices qui caractérisent les arguments qui sont déjà instanciés. Si le troisième argument d'un relation à trois arguments peut être instancié à condition

que les deux premiers arguments de cette relation le soient déjà alors la direction $\{1, 2\}$ est dite cohérente pour la relation. Une relation peut être utilisée avec toutes les directions qui sont cohérentes pour elle et les directions peuvent être inférées. Le filtrage de ML est utilisé pour discriminer les règles à appliquer et lorsqu'il n'est pas possible de discriminer, le mécanisme de "retour-arrière" de Prolog est encodé à l'aide de fermetures ML dans un mécanisme d'exceptions qui sont levées lorsqu'une règle ne permet plus de trouver une solution. Les spécifications non déterministes peuvent ainsi être extraites. Lorsque plusieurs fonctions (qui ont des sémantiques non équivalentes) peuvent correspondre à une relation et à une direction, l'une d'elles est choisie selon un critère fixé. A la notion de direction correspond celle de mode dans les articles plus récents [Berghofer and Nipkow 2000], ou de positions héritées/synthétisées dans [Attali and Parigot 1994]. Elle est relativement ancienne et largement utilisée, en particulier dans la communauté de la programmation logique. La dénomination de mode qui était déjà utilisée dans [Stärk 1994] est celle qui est restée dans l'usage. Dans d'autres approches telles que Centaur et RML, les modes n'apparaissaient pas car ils sont fixés syntaxiquement.

Une implantation du "$\mathbb{K}$ framework", l'outil $\mathbb{K}$ [Lazar et al. 2012], propose une solution complète pour décrire des langages de programmation et générer à partir de ces définitions des interprètes et des outils d'analyse de programmes. Il est par exemple possible, d'obtenir tous les comportements possibles d'un programme écrit dans un langage dont l'exécution n'est pas déterministe. Pour décrire un langage, on écrit une grammaire BNF annotée et un ensemble de règles qui peuvent être traduits vers LaTeX automatiquement. $\mathbb{K}$ supporte un système de modules qui rend possible l'utilisation d'une sémantique dans plusieurs outils. Il a été utilisé pour formaliser des langages complets tels que C [Ellison and Roşu 2012]. et Scheme, ainsi que dans des outils d'analyse de programmes.

Parmi les approches plus récentes, Spoofax [Kats and Visser 2010] est un outil consacré aux langages dédiés (DSL). Il comble le fossé qui existe entre les environnements de développement modernes, qui disposent de fonctionnalités avancées de détection des erreurs, de "refactoring", et d'aide à la saisie, et les outils généralement basiques créés spécifiquement pour les langages dédiés. Spoofax est une extension du célèbre environnement de développement intégré Eclipse. Mentionnons également XText [Eclipse.org 2013], un autre

plugin pour Eclipse, qui permet de créer des langages dédiés aussi bien que des langages de programmation généraux. Il est développé par la fondation Eclipse et se veut très simple d'accès. Il dispose de fonctionnalités avancées, et de langages dédiés pour l'écriture de systèmes de types [Bettini et al. 2012].

Dans une autre optique, Ott [Sewell et al. 2010] permet de faire un lien entre les outils pour la sémantique et les outils d'aide à la preuve. Ott propose à l'utilisateur de décrire la syntaxe d'un langage sous forme de grammaire et sa sémantique sous forme de règles d'inférence. Le tout est écrit dans un fichier texte qui doit respecter certaines conventions. À partir de ce fichier, Ott génère ensuite une spécification Coq, Twelf ou Isabelle/HOL correspondant au langage qui est décrit. Aucun mécanisme d'animation ou d'exécution des spécification ne semble, en revanche, disponible pour Ott.

Il est possible de faire le lien entre ces outils dédiés aux langages de programmation et les outils de preuves. Dans [Terrasse 1995], la traduction de spécifications Typol est par exemple assurée vers Coq, afin qu'il soit possible de raisonner sur ces spécifications.

3.3 Positionnement

Un écart important existe entre les outils d'aide à la preuve, présentés dans la première section, et les outils pour la sémantique des langages et pour les langages dédiés, présentés dans la seconde. Les outils d'aide à la preuve sont des outils généralistes, qui fournissent des moyens efficaces de raisonner. En revanche, ils ne bénéficient pas, pour la définition des langages, de la facilité d'accès ni de l'automatisation que l'on trouve par exemple dans des outils comme XText. Notre objectif, sans prétendre combler ce fossé, est de simplifier et rendre plus agréable l'utilisation d'outils d'aide à la preuve pour créer des langages et des outils, tout en gardant les fonctionnalités de raisonnement intéressantes des outils d'aide à la preuve. La solution que nous apportons consiste à rendre exécutables des spécifications inductives dans les outils d'aide à la preuve qui ne le permettent pas encore. Plusieurs travaux ont mis à disposition des fonctionnalités similaires dans des outils modernes tels que Isabelle/HOL et Coq pour supporter notamment l'exécution ou la génération d'interprètes à partir de spécifications, ou plus généralement l'exécution de spécifications. Notre approche

consiste à développer un mécanisme d'extraction, qui se veut général et qui est implanté dans deux outils : Coq et Focalize.

Dans le cadre de Coq, plus spécifiquement, nous étendons le processus d'extraction existant, afin d'obtenir un générateur de code fonctionnel efficace et complet.

Des outils similaires existent dans d'autres plateformes de preuve pour les méthodes formelles. L'atelier B [Abrial 1996] dispose de ProB [Leuschel and Butler 2008], un outil dont l'objectif est de valider les spécifications écrites par l'utilisateur afin d'augmenter la confiance de ce dernier dans les spécifications. Il permet notamment d'animer des spécifications B. En revanche, ProB ne génère pas de code exécutable. Un outil de simulation [Yang et al. 2012] permet de produire des instances exécutables de modèles Event-B, permettant de s'affranchir de certaines contraintes imposées par ProB comme l'obligation d'attribuer des valeurs explicites aux constantes. Un autre outil permet de générer du code C, C++, C# ou Java à partir de spécifications Event-B [Méry and Singh 2011].

Chapitre 4

Approche informelle

Ce chapitre a pour objectif de présenter le contexte de mes travaux de thèse et de détailler en quoi ils consistent sans rentrer dans les détails techniques, qui seront présentés dans le chapitre 5. Afin de présenter la génération de fonctions à partir de relations inductives, nous commençons par détailler ces deux notions, leurs points communs, leurs différences et les atouts de chacune dans la section 4.1. La section 4.2 montre comment il est possible de passer d'une approche à l'autre, manuellement dans un premier temps. L'automatisation du passage de la relation inductive à la fonction fait l'objet de cette thèse et des chapitres suivants.

4.1 Approches relationnelle et fonctionnelle

4.1.1 Relations inductives

Une définition des relations inductives a été donnée dans la section 2.1.4. L'approche relationnelle consiste à réaliser un développement en spécifiant de telles relations et en prouvant des théorèmes sur ces dernières. Nous avons présenté un exemple basique de relation inductive dans la figure 2.2.

Un autre exemple est présenté dans la figure 4.1 qui contient la sémantique naturelle associée au langage IMP (présentée dans le tableau 2.2) exprimée en Coq. La relation `exec` utilise les relations `eval` et `evalbool` qui spécifient respectivement l'évaluation d'une expression arithmétique (de type `expression`) et d'une expression booléenne (de type `boolean`) dans un état (de type `state`). La relation `exec` utilise également la fonction

`register` qui renvoie un état après y avoir ajouté une variable `x` avec la valeur `m`. Cet exemple montre que styles relationnel et fonctionnel peuvent être utilisés ensemble au sein d'une même spécification.

```
Inductive exec : command → state → state → Prop :=
 | exec_skip   : forall (s:state),
                   exec skip s s
 | exec_affect : forall (x:variable) (a:expression) (s:state)
                        (m:nat),
                   eval a s m → exec (affect x a) s (register s x m)
 | exec_seq    : forall (c0 c1:command) (s s' s'':state),
                   exec c0 s s'' → exec c1 s'' s' →
                   exec (seq c0 c1) s s'
 | exec_test1  : forall (b:boolean) (c0 c1:command) (s s':state),
                   evalbool b s true → exec c0 s s' →
                   exec (test b c0 c1) s s'
 | exec_test2  : forall (b:boolean) (c0 c1:command) (s s':state),
                   evalbool b s false → exec c1 s s' →
                   exec (test b c0 c1) s s'
 | exec_while1 : forall (b:boolean) (c:command) (s:state),
                   evalbool b s false → exec (while b c) s s
 | exec_while2 : forall (b:boolean) (c:command) (s s' s'':state),
                   evalbool b s true → exec c s s'' →
                   exec (while b c) s'' s' → exec (while b c) s s'.
```

FIGURE 4.1 – Spécification de l'exécution des commandes du langage IMP en Coq

4.1.2 Style fonctionnel

Lorsque l'utilisateur de Coq adopte le style fonctionnel, il écrit un programme comme le ferait un programmeur utilisant un langage fonctionnel, même si le langage fonctionnel dont il dispose est plus contraignant que d'autres langages fonctionnels comme OCaml ou Haskell par exemple. En effet, dans Coq toutes les fonctions doivent terminer et prouver leur terminaison constitue une obligation. De plus, les fonctions doivent être totales, ce qui impose en particulier que les filtrages soient systématiquement exhaustifs.

La différence majeure avec un programme classique se trouve dans l'ajout de spécifications et de théorèmes qui caractérisent le comportement du programme. Dans certains outils comme la plateforme d'analyse de code Frama-C [Cuoq et al. 2012] pour le langage C, les spécifications sont ajoutées au programme au moyen d'annotations. Dans Coq ou Twelf les spécifications et théorèmes sont écrits dans le même formalisme que le programme. Dans

tous les cas, les preuves sont réalisées avec un degré variable d'automatisation.

L'outil d'aide à la preuve Coq repose sur le calcul des constructions inductives (CIC) qui contient un langage fonctionnel avec les fonctions récursives et le filtrage sur les types inductifs. Plusieurs outils permettent de simplifier l'écriture des fonctions récursives : **`Fixpoint`**, **`Function`** [Barthe et al. 2006], et **`Program`** [Sozeau 2006]. **`Fixpoint`** est principalement utilisé pour les fonctions récursives structurelles mais c'est l'outil le plus simple d'utilisation. En effet, **`Fixpoint`** fait partie intégrante du CIC de Coq, alors que **`Function`** et **`Program`** sont des fonctionnalités introduites par des plugins qui définissent une nouvelle syntaxe, elle-même compilée vers le CIC. En contrepartie, ils offrent la possibilité d'écrire des programmes plus complexes (dont le schéma de récursion n'est pas structurel), ne laissant à l'utilisateur que quelques obligations de preuve. Pour pouvoir exécuter des fonctions qui ne terminent pas systématiquement ou bien des fonctions dont on ne veut pas montrer la terminaison, la technique qui consiste à ajouter à la fonction un argument de type entier naturel qui décroît à chaque appel récursif peut servir de pis-aller. Ainsi, la nouvelle fonction sera récursive structurelle et pourra être définie avec **`Fixpoint`**. On peut ainsi définir un interprète pour le langage IMP dont la sémantique est donnée dans le tableau 2.2, comme le montre la figure 4.2. Le nouveau type de retour de la fonction est un type `option`[1], et elle renvoie `None` si le compteur d'appels récursif atteint zéro. Ainsi, la fonction ne renverra un résultat que s'il est trouvé avec une profondeur d'appels inférieure à l'entier qui est passé en argument au premier appel de la fonction.

Lorsque la fonction est extraite vers OCaml en utilisant l'extraction native de Coq, il est possible de retrouver la fonction attendue en utilisant le code de la figure 4.3, où `inf` simule un entier infini.

En ce qui concerne les preuves dans le style fonctionnel, Coq offre la possibilité de raisonner par induction sur les fonctions (aussi appelée induction fonctionnelle). Le raisonnement par induction sur les fonctions développé dans [Barthe and Courtieu 2002] permet de générer automatiquement des schémas d'induction pour les fonctions et fournit une tactique pour automatiser les preuves par induction. L'induction est réalisée sur les

1. **`Inductive`** `option (A:`**`Type`**`) :` **`Type`** `:=`
`| Some : A →option A | None : option A.`

```
Fixpoint exec_imp (count:nat) (c:command) (s:state)
    : option state :=
  match (count, c) with
    | (O, _) ⇒ None
    | (S count, skip) ⇒ Some s
    | (S count, affect x a) ⇒ match eval_imp a s with
      | Some m ⇒ Some (register s x m)
      | None ⇒ None
    end
    | (S count, seq c0 c1) ⇒ match exec_imp count c0 s with
      | Some s' ⇒ exec_imp count c1 s'
      | None ⇒ None
    end
    | (S count, test b c0 c1) ⇒ match evalbool_imp b s with
      | Some true ⇒ exec_imp count c0 s
      | Some false ⇒ exec_imp count c1 s
      | None ⇒ None
    end
    | (S count, while b c0) ⇒ match evalbool_imp b s with
      | Some true ⇒ match exec_imp count c0 s with
        | Some s' ⇒ exec_imp count (while b c0) s'
        | None ⇒ None
      end
      | Some false ⇒ Some s
      | None ⇒ None
    end
  end.
```

FIGURE 4.2 – Interprète pour les commandes du langage IMP

```
let exec c s =
  let rec inf = S inf in
  match exec_imp inf c s with
    | None → assert false
    | Some s' → s'
```

FIGURE 4.3 – Utilisation de la fonction exec_imp en OCaml

chemins d'exécution de la fonction, et les preuves comportent alors un cas pour chacun des chemins d'exécution.

4.1.3 Avantages et inconvénients

Nous comparons les caractéristiques des deux approches, le but n'étant pas d'en déduire que l'une est meilleure que l'autre mais plutôt de montrer qu'elles ont toutes deux des avantages que l'autre n'a pas et qu'elles peuvent être complémentaires.

Tout d'abord, le style relationnel est souvent plus concis que le style fonctionnel. Cela est dû au fait que la syntaxe est plus compacte, mais surtout au fait qu'il autorise la définition de relations partielles, alors que les fonctions doivent être totales en Coq, ce qui oblige à énumérer tous les cas.

De plus, Coq ne dispose pas d'un mécanisme d'exceptions, et cela complique la définition de fonctions partielles. L'une des possibilités les plus simples pour écrire des fonctions partielles consiste à modifier le type de retour des fonctions en utilisant un type `option`. En écrivant des relations, on évite d'avoir à se préoccuper de la partialité.

Un autre avantage de l'approche relationnelle consiste à pouvoir s'abstraire des preuves de terminaison inhérentes à l'écriture de fonctions dans Coq.

En revanche, le style fonctionnel rend possible l'exécution de la fonction pour obtenir une valeur, alors que le style relationnel ne permet de réaliser aucun calcul. Il est uniquement possible de raisonner sur les relations.

Enfin, le raisonnement pour prouver des propriétés sur la fonction ou la relation inductive n'est pas le même. Il ne s'agit pas d'un avantage pour l'un ou l'autre, mais plus d'une question de goût car il est possible de raisonner par induction sur les fonctions, ce qui rend les preuves aussi simples que pour les relations inductives.

En conclusion, le style relationnel présente des avantages tant qu'on ne veut pas exécuter la spécification pour calculer des valeurs. Dans un contexte de programmation certifiée, l'exécution peut sembler fortuite, mais elle est pourtant essentielle afin d'animer et de valider une spécification. Sans exécution il est difficile de s'assurer qu'une spécification traduira bien le comportement attendu. On peut alors prendre le risque d'effectuer de nombreuses

preuves de propriétés sur une spécification fausse. Il n'y a donc pas d'approche idéale et les spécifications Coq comporte généralement à la fois des fonctions et des définitions inductives. Une approche souvent adoptée consiste à écrire deux versions du programme, l'une avec des relations, l'autre avec des fonctions, et d'écrire une preuve d'équivalence entre les deux [Blazy and Leroy 2009; Campbell 2012].

Notons que le style fonctionnel peut parfois se rapprocher du style relationnel, comme c'est le cas dans Agda2 qui permet de décrire des sémantiques opérationnelles en utilisant des fonctions même lorsque ces dernières sont partielles [Danielsson 2012].

4.2 Passer d'une approche à l'autre

Nous avons vu que les deux approches exposées précédemment sont parfois utilisées conjointement. L'utilisateur, du moins dans le cas de Coq, doit alors écrire deux fois le code, mais aussi la preuve d'équivalence entre les deux versions. Nous nous intéressons ici aux moyens que nous pouvons mettre en œuvre pour passer d'une version à l'autre, manuellement dans un premier temps, puis automatiquement dans les chapitres suivants. Bien entendu, nous ne considérons pas ici le cas général. Il est parfois impossible d'obtenir une fonction à partir d'une relation inductive, et il est parfois inutile d'essayer de construire une relation inductive à partir d'une fonction. Nous restreignons volontairement notre approche au cas où l'on peut extraire des fonctions ne renvoyant qu'un seul résultat à partir de relations, et ce de manière déterministe.

4.2.1 De la relation inductive à la fonction

Dans certains cas, la définition d'une relation inductive peut contenir plusieurs comportements calculatoires. Prenons l'exemple de la relation `add` (figure 2.2). Il est possible d'écrire une fonction à trois arguments n_1, n_2 et n_3 qui renvoie `true` lorsque la relation est définie et `false` dans les autres cas. Mais on peut aussi écrire d'autres fonctions, à deux arguments, comme l'addition de deux entiers qui calcule n_3 à partir de n_1 et n_2 (`add12` dans la figure 4.4), ou encore la soustraction qui calcule n_1 à partir de n_2 et n_3 (`add23` dans la figure 4.4).

```
Fixpoint add12 (n1:nat) (n2:nat) : nat :=
  match n2 with
    | O ⇒ n1
    | S n ⇒ S (add12 n1 n)
  end.

Fixpoint add23 (n2:nat) (n3:nat) : option nat :=
  match (n2, n3) with
    | (O, n3') ⇒ Some n3'
    | (S n2', S n3') ⇒ add23 n2' n3'
    | _ ⇒ None
  end.
```

FIGURE 4.4 – Fonction d'addition add12, et de soustraction add23

Afin d'identifier un comportement calculatoire dans une relation inductive, nous utilisons la notion de mode. Un mode est un ensemble d'indices qui identifient les arguments de la relation que l'on veut utiliser comme argument d'une fonction. Les arguments restants sont alors calculés par cette fonction. Si une fonction est extraite à partir de la relation inductive, on parle de mode d'extraction de la fonction. Si l'on reprend les exemples précédents, la fonction d'addition peut être extraite de la relation `add` avec le mode $\{1, 2\}$ et la fonction de soustraction avec le mode $\{2, 3\}$. Nous nommons généralement les fonctions extraites en concaténant le nom de la relation et le mode d'extraction : `add12`, `add23`. Le mode $\{1, 2, 3\}$ peut également être utilisé pour extraire une fonction à trois arguments. Lorsque tous les arguments sont utilisés comme entrée pour la fonction, cette dernière renvoie un booléen. On parle alors de mode d'extraction complet.

Il est également possible de considérer des exemples plus réalistes. Nous avons déjà présenté dans la section précédente la relation inductive `exec` et la fonction `exec_imp` qui concernent toutes deux la sémantique du langage IMP. La fonction `exec_imp` a été extraite à partir de la relation `exec` avec le mode $\{1, 2\}$.

L'étape restante concerne l'écriture des preuves de correction et de complétude de la fonction par rapport à la relation inductive. Nous détaillons ici les preuves manuelles pour le cas très simple de l'addition dont la fonction est présentée dans la figure 4.4. La génération automatique de ces preuves fait l'objet de la dernière section du chapitre 7.

Le lemme de correction, prouvé dans la figure 4.5, assure que tous les résultats calculés

par la fonction sont ceux qui sont définis dans la relation inductive. La preuve est réalisée par induction sur la fonction `add12`. Pour cela, on commence par générer un schéma d'induction pour la fonction, puis on applique ce dernier et on termine avec une preuve pour chacun des deux cas d'induction.

```
Functional Scheme add12_ind := Induction for add12 Sort Prop.

Lemma add12_sound :
  forall n1 n2 n3, add12 n1 n2 = n3 → add n1 n2 n3.
Proof.
intros. subst n3. apply add12_ind. (* induction scheme application *)
intros. apply add0.                (* case 1 *)
intros. apply addS. apply H.       (* case 2 *)
Qed.
```

FIGURE 4.5 – Preuve de correction pour la fonction add12

La preuve de complétude est réalisée dans la figure 4.6. Elle montre que partout où la relation inductive est définie, la fonction renvoie un résultat et un résultat correct.

```
Lemma add12_complete :
  forall n1 n2 n3, add n1 n2 n3 → add12 n1 n2 = n3.
Proof.
induction 1.                       (* induction scheme application *)
simpl. reflexivity.                (* case 1 *)
subst. reflexivity.                (* case 2 *)
Qed.
```

FIGURE 4.6 – Preuve de complétude pour la fonction add12

Mes travaux de thèse sont essentiellement motivés par le fait d'automatiser la démarche qui vient d'être présentée : la génération de fonctions à partir de relations inductives ainsi que la génération des lemmes et des preuves associées à ces fonctions. Cependant, l'exemple qui précède ne concerne que la génération de fonctions Coq à partir de spécifications elles-mêmes écrites en Coq. Mes travaux de thèse concernent plus généralement l'extraction de programmes fonctionnels à partir de relations inductives, pour différents langages cibles et sources. Pour cela, un schéma de compilation général a été établi, qui se décline pour chaque langage. Dans les chapitres suivants, nous étudions :

— la génération de fonctions OCaml et Haskell à partir de Coq (chapitre 6),

— la génération de fonctions Coq à partir de Coq, ainsi que le preuves (chapitre 7),

— la génération de fonctions Focalize et OCaml à partir de Focalize (chapitre 8).

Le schéma d'extraction général, commun à toutes ces implantations, est présenté et formalisé dans le chapitre 5. Il décrit la transformation d'une relation inductive vers une fonction exprimée dans un langage intermédiaire commun.

4.2.2 De la fonction à la relation inductive

Nous ne détaillons pas ici cet aspect, car il sort du cadre de cette thèse. Il est néanmoins possible de générer le graphe d'une fonction Coq en mimant la définition fonctionnelle. Le plugin Coq "Function" [Barthe et al. 2006] s'appuie sur ce graphe pour construire la fonction et commence donc par construire ce dernier. Par exemple, la figure 4.7 montre la définition d'une fonction avec **Function**, et le graphe qui est construit lors de cette définition. Bien entendu, la relation obtenue n'est pas toujours celle que l'on écrirait manuellement, mais sur de petits exemples, on retrouve une relation proche de celle que l'on aurait écrite naturellement.

```
Function add12 (n1:nat) (n2:nat) : nat :=
  match n2 with
    | 0 ⇒ n1
    | S n ⇒ S (add12 n1 n)
  end.
Print R_add12.
```

La commande `Print` affiche le résultat suivant :

```
Inductive R_add12 (n1 : nat) : nat → nat → Type :=
    R_add12_0 : forall n2 : nat, n2 = 0 → R_add12 n1 0 n1
  | R_add12_1 : forall n2 n : nat,
                n2 = S n →
                forall res : nat,
                R_add12 n1 n res → R_add12 n1 (S n) (S res).
```

FIGURE 4.7 – Relation inductive générée à partir de la fonction add12

Le mécanisme de **Function** peut être utilisé avec des fonctions extraites par notre outil. Il suffit, pour s'en convaincre de copier le code de la fonction générée et de créer une nouvelle définition utilisant **Function**.

En revanche, le contraire qui consiste à utiliser notre outil pour générer du code à partir

de relations générées par **`Function`** n'est pas présenté ici car il reste beaucoup de travail pour donner aux graphes de **`Function`** une forme qui soit acceptée par notre outil.

Chapitre 5

De la relation inductive au programme fonctionnel

Notre travail fait suite à une première implantation de l'extraction des relations inductives réalisée au sein de l'outil d'aide à la preuve Coq, qui a fait l'objet d'une réécriture complète. Nous commençons par définir formellement la forme des relations inductives dans la section 5.1, avant de résumer cette première approche dans la section 5.3. Cette présentation permet d'exposer les concepts essentiels de l'extraction des relations inductives. Après cela, nous formalisons notre approche, en commençant par justifier une nouvelle représentation de données intermédiaire pour les relations inductives dans la section 5.4, pour laquelle nous donnons un schéma de compilation dans la section 5.5. Cette représentation de données et ses invariants seront formalisés dans la section 5.6. Puis, nous présentons la génération de code fonctionnel en utilisant la nouvelle représentation intermédiaire dans les sections 5.7 et 5.8. Enfin, nous exposons quelques extensions dans la section 5.9.

5.1 Formalisation des relations inductives

Nous avons donné la définition d'une relation inductive dans la section 2.1.4. Nous précisons ici la forme exacte attendue d'une relation inductive dont on souhaite extraire des fonctions. Les relations inductives que nous considérons sont un sous-ensemble des inductifs que l'on peut définir en Coq à l'aide du mot clé **`Inductive`**, suffisamment restreint pour que l'approche puisse être généralisée à d'autres outils de preuve.

Très simplement, nous interdisons les paramètres dans les inductifs, et nous restreignons les arguments de la relation inductive à des types de données inductifs. De plus les termes qui sont utilisés dans la définition de la relation inductive doivent être des constructeurs de types de données inductifs.

5.1.1 Types de données inductifs

Les types de données inductifs que nous considérons sont ceux définis par le CIC de Coq, réduits à l'essentiel. Nous limitons les définitions dans **`Set`**, et un type de données inductif est réduit à une liste de constructeurs nommés où chaque constructeur possède une liste d'arguments qui ont chacun pour type un type de données inductif. Nous ne considérons pas les types paramétrés afin de simplifier la formalisation et pour qu'elle soit valable pour d'autres outils que Coq. Néanmoins, certaines restrictions sont levées par la suite dans le cas de l'extraction au sein de Coq, en tant qu'extensions.

5.1.2 Relations inductives

Les inductifs que nous considérons ici sont des relations dans **`Prop`**, dont tous les arguments ont pour type un type de données inductif au sens défini précédemment. Ils ont la forme suivante :

$\mathsf{Ind}(d : \tau, \Gamma)$

où d est le nom de la relation inductive, τ son type et Γ le contexte dans lequel se trouvent les constructeurs (leur nom accompagné de leur type respectif). Dans cette notation, une relation inductive est définie par un type et un ensemble de constructeurs nommés. En revanche, deux restrictions apparaissent par rapport aux inductifs de Coq : il ne peut pas y avoir de paramètres[1] ni de définitions mutuellement récursives. De plus, les types dépendants, l'ordre supérieur et les arguments propositionnels ne sont pas autorisés dans le type de la définition inductive. Plus précisément, cela signifie que τ a la forme suivante :

$\tau_1 \rightarrow \ldots \tau_i \rightarrow \ldots \tau_n \rightarrow \mathsf{Prop}$

où τ_i est le nom d'un type de données inductif, et ne contient donc ni produit ni types dépendants, pour $i = 1 \ldots n$. De plus, on suppose que les types des constructeurs sont sous

1. Le support des paramètres nécessiterait de les ajouter en argument de la fonction extraîte.

forme prénexe, et ne contiennent ni dépendance aux variables liées ni ordre supérieur. Le type des constructeurs est alors le suivant :

$\forall x_1 : X_1, \ldots x_i : X_i, \ldots x_n : X_n . T_1 \rightarrow \ldots T_j \rightarrow \ldots T_m \rightarrow (d\ t_1\ \ldots\ t_p)$

où $x_i \notin X_l$, avec $l > i$, X_i est un type de données inductif, et t_k est un terme, avec $k = 1 \ldots p$. Par la suite, nous nommons les termes T_j les prémisses du constructeur et le terme $(d\ t_1\ \ldots\ t_p)$ sa conclusion. On impose aussi que T_j soit une relation inductive complètement appliquée (elle a autant d'arguments que son arité le permet), de la forme :

$d_j\ t_{j1}\ \ldots\ t_{jp_j}$

où d_j est une relation inductive (qui peut être différente de d), t_{jk} est un terme, avec $k = 1 \ldots p_j$, et p_j est l'arité de d_j.

De plus, le terme t_i est une variable ou un constructeur (d'un type de données inductif) complètement appliqué $c\ t_1\ \ldots\ t_{p_c}$, où p_c est l'arité de c.

Un type inductif qui vérifie les conditions ci-dessus est nommé relation inductive dans la suite. Dans ce chapitre nous nous limitons à ces relations inductives.

5.2 Définitions et notations

Dans la suite, nous utilisons le nom des constructeurs pour référencer ces derniers. Pour accéder à la définition du constructeur C, on utilisera la notation $\Gamma(C)$. Nous introduisons également la notation $P(C)$ pour faire référence à l'ensemble des prémisses du constructeur nommé C, ainsi que la notation $P(C, i)$ pour faire référence à la i^{me} prémisse de ce constructeur.

Afin de manipuler plus facilement les termes et les modes, nous définissons un ensemble de fonctions dont l'objectif est de calculer les entrées et les sorties d'une relation inductive pour un mode d'extraction donné. Pour rappel, le mode est un ensemble d'indices qui désignent les arguments de la relation qui seront les arguments de la fonction extraite. Le mode ne précise donc pas d'ordre pour les arguments de la fonction extraite à partir d'une relation inductive, mais seulement le numéro des arguments. Les arguments apparaitront toujours dans le même ordre dans la fonction et la relation inductive. La fonction in renvoie le tuple qui contient les arguments qui correspondent aux

indices du mode. Par exemple, $\mathsf{in}(add\ (S\ n)\ m\ (S\ p), \{1,2\}) = (S\ n, m)$. La fonction out renvoie le terme correspondant à l'argument restant lorsqu'il n'y en a qu'un. Par exemple, $\mathsf{out}(add\ (S\ n)\ m\ (S\ p), \{1,2\}) = S\ p$. Quand le mode est complet, c'est-à-dire que tous les arguments de la relation inductive sont des entrées, la fonction out renvoie le terme *true*. S'il reste plusieurs sorties pour la fonction extraite, out renvoie un tuple contenant les termes correspondant aux sorties. Les sorties seront alors ordonnées dans le tuple dans le même ordre que les arguments dans la relation inductive. De plus, nous définissons également invars et $\mathsf{outvars}$ qui renvoient les variables qui apparaissent dans les arguments respectivement en entrée et en sortie. Toutes ces fonctions sont regroupées dans la définition suivante :

Définition 1 (Manipulation des termes relativement au mode d'extraction)
Pour une relation inductive d, les termes $t_1 \ldots t_{p_d}$, et un mode m, nous définissons les fonctions suivantes :

$$\mathsf{in}(d\ t_1\ \ldots\ t_{p_d}, m) \triangleq (t_{i_1},\ \ldots,\ t_{i_m})\ \textit{où}\ m\ \textit{est}\ \{i_1,\ \ldots,\ i_m\}$$

$$\mathsf{invars}(d\ t_1\ \ldots\ t_{p_d}, m) \triangleq \mathsf{variables}(\mathsf{in}(d\ t_1\ \ldots\ t_{p_d}, m))$$

$$\mathsf{out}(d\ t_1\ \ldots\ t_{p_d}, m) \triangleq \begin{cases} \textit{si}\ \exists! j \in 1..p_d, j \notin \{i_1,\ \ldots,\ i_m\}\ \textit{alors}\ t_j \\ \textit{si}\ \exists j_1..j_n\ \textit{(suite croissante)}, n > 1, \forall k, j_k \notin \{i_1,\ \ldots,\ i_m\} \\ \qquad \textit{alors}\ (t_{j_1},\ \ldots,\ t_{j_n}) \\ \textit{sinon}\ true \end{cases}$$

$$\textit{où}\ m\ \textit{est}\ \{i_1,\ \ldots,\ i_m\}$$

$$\mathsf{outvars}(d\ t_1\ \ldots\ t_{p_d}, m) \triangleq \mathsf{variables}(\mathsf{out}(d\ t_1\ \ldots\ t_{p_d}, m))$$

où $\mathsf{variables}(t)$ *renvoie l'ensemble des variables qui sont présentes dans le terme t, qui peut éventuellement être un tuple.*

Nous définissons également la fonction rel *telle que :* $\mathsf{rel}(d\ t_1\ \ldots\ t_{p_d}) = d$.

Nous introduisons également un environnement global $\mathcal{M}$, qui contient les modes d'extraction pour toutes les relations inductives utilisées dans la relation inductive que l'on est en train d'extraire. Le mode d'extraction pour la relation inductive d est obtenu avec $\mathcal{M}(d)$. Nous ajoutons par commodité la notation suivante : si $T_i = d_i\ t_{i1}\ \ldots\ t_{ip_i}$, alors $\mathcal{M}(T_i) = \mathcal{M}(d_i)$. Par la suite, d désignera toujours la relation inductive que l'on est en train d'extraire et $m = \mathcal{M}(d)$ son mode d'extraction. Les autres relations inductives ou les autres modes seront toujours indicés.

5.3 Schéma d'extraction de base

Cette section a pour objectif de résumer l'approche utilisée pour implanter le premier extracteur de relations inductives pour Coq [Delahaye et al. 2007]. Tout comme dans l'approche actuelle, que nous détaillons après cette section, l'ensemble des modes d'extraction doit être fourni par l'utilisateur. La notion de relations inductive est la même dans les deux approches. Une fois les modes fournis, l'extraction comporte deux étapes. La première consiste à vérifier que l'extraction est possible en réalisant une analyse de flot de données qui détermine s'il est possible de calculer les variables utilisées dans les sorties de la fonction extraite. La seconde étape est l'extraction de code proprement dite, qui peut éventuellement échouer.

On ne s'intéresse dans cette thèse qu'aux spécifications qui permettent d'extraire des fonctions qui renvoient un seul résultat pour des entrées données et ce de manière déterministe. Dans le cas général, on ne sait pas extraire toutes les spécifications qui peuvent conduire à des fonctions déterministes, ni même vérifier le déterminisme d'une spécification. L'approche qui est employée ici consiste à se limiter aux spécifications dont les entrées (par rapport au mode d'extraction) des conclusions des constructeurs ne se recouvrent pas deux à deux. Ainsi, le premier filtrage dans la fonction générée permet d'assurer le choix du constructeur qui sera utilisé pour calculer la sortie de la fonction pour des entrées données. Permettre l'extraction d'autres types de spécifications plus complexes, notamment des spécifications dont les entrées des conclusions peuvent se recouvrir, est l'un des enjeux de cette thèse.

Le mode d'extraction pour chaque relation inductive à extraire est donné dans une nouvelle commande Coq qui déclenche l'extraction des fonctions. Dans l'implantation de 2007, il n'était pas possible de générer des fonctions Coq, mais uniquement des fonctions OCaml (ou Haskell) dans un fichier séparé.

Plus précisément, la relation inductive est compilée en un une fonction du langage MiniML, langage intermédiaire de l'extraction native de Coq [Letouzey 2002]. Ce mécanisme prend alors le relai pour produire le code OCaml ou Haskell.

Dans la suite, nous ne détaillons pas complètement la formalisation décrite dans l'ar-

ticle original, mais nous abordons les points nécessaires à la compréhension de la suite du chapitre. De plus la description de la première version peut permettre au lecteur de se faire une idée des progrès réalisés dans la nouvelle version. L'analyse de cohérence de mode est décrite en détail car elle n'a que très peu évolué. La génération de code, en revanche, est décrite brièvement car elle suit maintenant un processus plus complexe.

5.3.1 Analyse de cohérence de mode

L'analyse de cohérence de mode permet de vérifier que seules les variables liées seront utilisées dans la fonction extraite. C'est une analyse de flot de données réalisée constructeur par constructeur, pour une relation inductive dont on considère l'extraction avec un mode donné. Une relation inductive peut utiliser d'autres relations inductives, il est nécessaire de fournir également le mode d'extraction de ces relations inductives car l'analyse de cohérence de mode les concerne également.

Un mode m_i est dit cohérent pour une relation inductive d_i si il est cohérent pour tous ses constructeurs Γ. Un mode m_i est dit cohérent pour un constructeur si, en considérant sa conclusion t et ses prémisses ordonnées $p_1,\ \ldots,\ p_n$:

- Les variables utilisées dans p_1 ($\mathsf{invars}(p_1, \mathcal{M}(p_1))$) sont définies par la conclusion ($\mathsf{invars}(t, \mathcal{M}(d))$).
- Les variables utilisées dans une prémisse p_j sont définies par la conclusion ou une prémisse $p_{j'}$ avec $j' < j$ ($\mathsf{outvars}(p_{j'}, \mathcal{M}(p_{j'}))$).
- Les variables utilisées dans la conclusion ($\mathsf{outvars}(t, \mathcal{M}(d))$) sont définies dans la conclusion ou l'une des prémisses.

L'ordre des prémisses n'a pas d'importance dans la définition d'une relation inductive. Dans certains cas, il est nécessaire d'ordonnancer les prémisses dans un ordre différent de celui qu'a donné l'utilisateur. L'analyse de cohérence de mode est pour cela réalisée sur toutes les permutations de prémisses jusqu'à en trouver une pour laquelle le mode choisi est cohérent pour chacun des constructeurs de la relation inductive. L'analyse échoue si une telle permutation n'existe pas.

Plus formellement, on considère un constructeur C, ayant la forme suivante :

$\forall x_1 : X_1, \ldots x_i : X_i, \ldots x_n : X_n . T_1 \rightarrow \ldots T_j \rightarrow \ldots T_m \rightarrow (d\ t_1\ \ldots\ t_p)$

Et on considère également un ensemble de variables S_j, pour $j = 0..m$. L'analyse de cohérence de mode pour une permutation π des prémisses du constructeur C est vérifiée si les quatre équations suivantes le sont :

1. $S_0 = \mathsf{invars}(d\ t_1\ \ldots\ t_p, m_d)$
2. $\mathsf{invars}(T_{\pi(j)}, \mathcal{M}(T_{\pi(j)})) \subseteq S_{j-1}, \text{avec } 1 \leq j \leq m$
3. $S_j = S_{j-1} \bigcup \mathsf{outvars}(T_{\pi(j)}, \mathcal{M}(T_{\pi(j)})), \text{avec } 1 \leq j \leq m$
4. $\mathsf{outvars}(d\ t_1\ \ldots\ t_p, m_d) \subseteq S_m$

L'ensemble S_0 dénote l'ensemble des variables connues initial et S_j l'ensemble des variables connues après l'exécution de la $\pi(j)^{ime}$ prémisse. La première condition traduit que les variables qui sont des entrées de la conclusion du constructeur C doivent être connues. La deuxième condition permet de vérifier que l'exécution d'une prémisse $T_{\pi(j)}$ n'aura lieu que si toutes les variables qui apparaissent dans ses entrées (avec le mode d'extraction $\mathcal{M}(T_{\pi(j)})$) sont calculables. D'après la troisième condition, tous les arguments de de la prémisse $T_{\pi(j)}$ sont connus après son exécution. Enfin, le mode m_d est dit cohérent pour C si tous les arguments de la conclusion sont connus après l'exécution de toutes les prémisses.

Si on prend l'exemple de l'addition, avec le constructeur `addS` on a :

1. $S_0 = \mathsf{invars}(\texttt{add n (S m) (S p)}, \{1,2\}) = \{n, m\}$
2. $\mathsf{invars}(\texttt{add n m p}, \{1,2\}) = \{n, m\} \subseteq S_0$
3. $S_1 = S_0 \bigcup \mathsf{outvars}(\texttt{add n m p}, \{1,2\}) = \{n, m, p\}$
4. $\mathsf{outvars}(\texttt{add n (S m) (S p)}, \{1,2\}) \subseteq S_1$

Dans ce cas, l'analyse de cohérence de mode est vérifiée.

5.3.2 Génération de code MiniML

Une fois l'analyse de cohérence de mode effectuée avec succès, on sait que les valeurs de toutes les variables peuvent être calculées (si la fonction termine). Le code fonctionnel peut alors être généré. La génération de code est effectuée vers un langage fonctionnel comportant la récursion et le filtrage avec gardes. Ce langage n'est pas compatible avec

le langage utilisé par le mécanisme d'extraction natif de Coq, ce qui empêche une intégration harmonieuse de l'extraction à partir de relations inductives au sein de Coq. La solution adoptée fut de modifier temporairement le langage MiniML de l'extraction native en y ajoutant les gardes. La production de code est réalisée en deux temps. Un premier filtrage sur l'ensemble des arguments de la fonction est construit avec un cas pour chaque constructeur de la relation inductive. Chaque motif de filtrage est produit à partir de la conclusion du constructeur. Ensuite, pour chaque constructeur, chaque prémisse produit un filtrage ou un "let-in". La valeur renvoyée, spécifique à chaque constructeur, est la sortie de sa conclusion si le mode d'extraction n'est pas complet et `true` si le mode est complet.

5.3.3 Principale limitation

La principale limitation imposée par cette première approche est l'heuristique choisie pour s'assurer qu'une relation inductive associée à un mode d'extraction est déterministe, c'est-à-dire que la fonction extraite à partir de la relation selon ce mode ne renvoie qu'un seul résultat pour des entrées données. Le fait que les entrées des conclusions des constructeurs ne se recouvrent pas est un critère très restrictif. Une optimisation possible est néanmoins mentionnée dans [Delahaye et al. 2007] pour relaxer cette contrainte. Aucun algorithme, en revanche, n'est donné pour produire le résultat présenté dans la figure 5.1. Dans cet exemple, la spécification est celle de la boucle "while" d'un mini langage impératif. Les conclusions des constructeurs `while1` et `while2` ont des entrées qui se recouvrent mais on peut néanmoins en extraire du code déterministe.

5.4 Représentation de données pour l'extraction

Le besoin d'une nouvelle façon de représenter les relations inductives est né de la principale amélioration que nous apportons par rapport à la première approche décrite dans la section précédente. Cette extension est essentielle si l'on veut pouvoir traiter des relations inductives plus réalistes.

L'exemple de la boucle de la figure 5.1 fait clairement apparaitre qu'il est possible de traiter davantage de spécifications en fusionnant, dans la fonction générée, des conclu-

Spécification de la relation inductive :

```
Inductive exec : store → command → store → Prop := ...
  | while1 : forall (s s1 s2 : store) (b : expr) (c : command),
      (eval s b true) → (exec s c s1) →
      (exec s1 (While b do c) s2) → (exec s (While b do c) s2)
  | while2 : forall (s : store) (b : expr) (c : command),
      (eval s b false) → (exec s (While b do c) s).
```

où `store` est le type des contextes d'évaluation, `command` est le type des instructions, `While` est le constructeur pour les boucles et `expr` le type des expressions. La relation inductive `eval` définit l'évaluation des expressions.

Proposition de fonction extraite :

```
let rec exec s c = match s, c with ...
  | s, While (b, c) →
    (match eval s b with
       | true →
         let s1 = exec s c in
         let s2 = exec s1 (While (b, c)) in s2
       | false → s)
```

FIGURE 5.1 – Sémantique de `While`

sions dont les entrées sont identiques. On utilise alors des prémisses (`eval s b true` et `eval s b false` dans l'exemple) pour déterminer quel constructeur sera utilisé pour calculer la valeur renvoyée pour des entrées données. On peut également fusionner des prémisses lorsque cela est nécessaire, dans le cas où deux constructeurs possèdent non seulement des conclusions, mais aussi certaines prémisses identiques (`(exec s c s1)` dans la figure 5.2). Il peut en effet être nécessaire de fusionner des prémisses si l'ordre d'exécution imposé par l'analyse de cohérence de mode place les prémisses en question avant celles qui permettent de distinguer les constructeurs. Un autre exemple est donné plus loin dans ce chapitre, dans la figure 5.14 (pour les constructeurs `Search_inf` et `Searchinf_nf` avec la prémisse `compare n m Inf`).

Le processus d'extraction est alors réalisé en deux étapes. La première consiste à ordonner les prémisses et à choisir quelles conclusions et prémisses seront fusionnées. La seconde permet de générer du code fonctionnel. Pour faire le lien entre ces deux étapes, nous introduisons une représentation intermédiaire arborescente que nous nommons Reltree. Les Reltree sont indépendants du langage cible et permettent de mémoriser l'ordre des prémisses ainsi que les fusions opérées entre conclusions ou entre prémisses.

Spécification de la relation inductive :

```
Inductive exec : store → command → store → Prop := ...
  | repeat1 : forall (s s1 : store) (b : expr) (c : command),
      (exec s c s1) → (eval s1 b true) →
      (exec s (Repeat c until b) s1)
  | repeat2 : forall (s s1 s2 : store) (b : expr) (c : command),
      (exec s c s1) → (eval s1 b false) →
      (exec s1 (Repeat c until b) s2) → (exec s (Repeat c until b) s2).
```

où `store` est le type des contextes d'évaluation, `command` est le type des instructions, `Repeat` est le constructeur pour les boucles de type `expr`, qui est le type des expressions. La relation inductive `eval` définit l'évaluation des expressions.

Proposition de fonction extraite :

```
let rec exec s c = match s, c with ...
  | s, Repeat (c, b) →
    let s1 = exec s c in ( match eval s1 b with
      | true → s1
      | false → exec s1 (Repeat (c, b)) )
```

FIGURE 5.2 – Sémantique de `Repeat`

L'objectif de cette section est de présenter de manière informelle les Reltree, qui sont formalisés dans la section 5.6. La section 5.4.1 propose un exemple de cette représentation et la section 5.4.2 explicite les liens qui existent entre la relation inductive, le Reltree et le code généré.

5.4.1 Exemple d'extraction utilisant un Reltree

Considérons la spécification de la figure 5.3, qui définit la recherche d'un chemin dans un arbre binaire de recherche. La relation inductive `compare` permet de comparer deux entiers naturels, et la relation `search` décrit le chemin qui mène à un entier naturel donné dans un arbre binaire de recherche. La spécification ci-dessous n'est pas très élégante, car dans le cas où l'entier n'est pas trouvé dans l'arbre, le chemin ne sera pas forcement `NotFound` mais un chemin qui se termine par `NotFound`. Une meilleure version nécessiterait d'ordonnancer les constructeurs, ce que nous ferons plus tard (dans la section 5.9.4).

Appliquons maintenant le processus d'extraction aux deux relations inductives `compare` et `search` avec le mode $\{1, 2\}$. On obtient alors deux fonctions `compare12` et `search12`. Rappelons que l'implantation actuelle de notre plugin d'extraction permet d'extraire au choix du code OCaml ou du code Coq. Nous pouvons obtenir deux fonctions OCaml. La

```
Inductive abr : Set :=
  | Empty : abr
  | Node : abr → nat → abr → abr.

Inductive comp_nat : Set := | Inf | Sup | Eq.

Inductive path: Set :=
 | NotFound | EndPath
 | Left : path → path | Right : path → path.

Inductive compare : nat → nat → comp_nat → Prop :=
  | Compare_true : compare 0 0 Eq
  | Compare_inf : forall n, compare 0 (S n) Inf
  | Compare_sup : forall n, compare (S n) 0 Sup
  | Compare_rec : forall n m c, compare n m c →
      compare (S n) (S m) c.

Inductive search : abr → nat → path → Prop :=
  | Search_empty : forall n, search Empty n NotFound
  | Search_found : forall n m t1 t2, compare n m Eq →
      search (Node t1 m t2) n EndPath
  | Search_inf : forall n m t1 t2 b, search t1 n b →
      compare n m Inf → search (Node t1 m t2) n (Left b)
  | Search_sup : forall n m t1 t2 b, search t2 n b →
      compare n m Sup → search (Node t1 m t2) n (Right b).
```

FIGURE 5.3 – Recherche d'un chemin dans un arbre binaire de recherche

figure 5.4 présente la fonction `search12`.

```
let rec search12 p1 p2 = match (p1, p2) with
  | (Empty, n) → NotFound
  | (Node (t1, m, t2), n) → (match compare12 n m with
     | Eq → EndPath
     | Inf →
       (match search12 t1 n with
        | b → Left b)
     | Sup →
       (match search12 t2 n with
        | b → Right b))
```

FIGURE 5.4 – Extraction de la relation inductive search avec le mode $\{1, 2\}$ vers OCaml

Pour obtenir ce code, on utilise le Reltree de la figure 5.5, qui se lit de gauche à droite. Un autre Reltree est produit pour la relation `compare` mais il n'est pas montré ici. Un Reltree n'est pas un arbre, mais un ensemble d'arbres, c'est-à-dire une forêt. Pour une relation inductive qui ne nécessite aucune fusion, il y a autant d'arbres que de constructeurs. Les nœuds dont les coins sont droits contiennent des termes issus de conclusions alors que ceux dont les coins sont arrondis contiennent des termes issus de prémisses. Les conclusions sont donc placées à la racine et dans les feuilles des arbres. Les autres nœuds correspondent à des prémisses. De plus, nous masquons certains arguments des conclusions par des tirets afin d'accroitre la lisibilité. Les arguments masqués sont ceux qui ne sont pas utilisés, c'est-à-dire les sorties quand les conclusions sont des racines (qui servent à construire le premier filtrage de la fonction) et les entrées quand il s'agit des feuilles (qui servent à construire les sorties de la fonction).

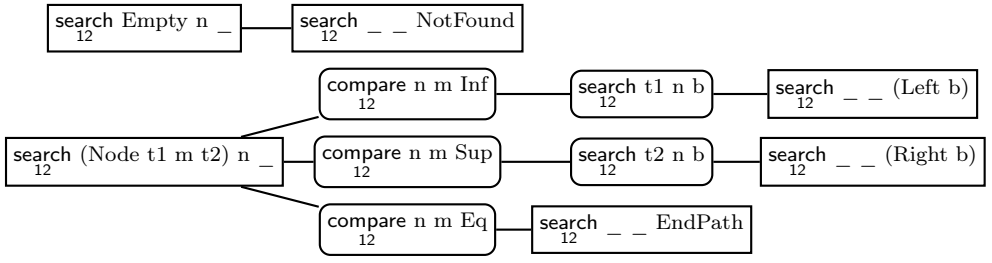


FIGURE 5.5 – Reltree de la relation inductive `search` en mode $\{1, 2\}$

5.4.2 Liens entre Reltree, relation inductive et code généré

Graphiquement, les Reltree sont très proches de la relation inductive. Il est possible de retrouver les différents constructeurs de la relation inductive à partir du Reltree, à l'exception de leur nom. Considérons l'ensemble des chemins qui permettent de traverser les arbres que contient le Reltree, depuis la racine jusqu'à une feuille. À chacun de ces chemins correspond un constructeur dans la relation inductive. Certaines conclusions ou prémisses ayant été fusionnées, un nœud peut être utilisé pour représenter des conclusions ou prémisses appartenant à des constructeurs différents. Ces prémisses ou conclusions sont alors identiques dans les différents constructeurs à renommage de variables près.

Le Reltree étant assez proche de la fonction du point de vue de l'exécution, la génération de code devient aisée une fois le Reltree trouvé. En effet, le premier filtrage, sur les arguments de la fonction est construit en utilisant un motif pour chaque racine du Reltree. Ensuite, à chaque groupe de nœuds qui partagent le même père correspond un filtrage. Les termes sur lesquels s'effectuent le filtrage sont déterminés à partir des entrées (elles sont toujours les mêmes pour les nœuds qui partagent un même père) et les sorties sont les motifs de filtrage (elles sont toujours différentes pour les nœuds qui partagent un même père). Dans notre exemple, les trois nœuds `compare n m Inf`, `compare n m Sup` et `compare n m Eq` partagent le même père et sont utilisés pour construire le filtrage suivant dans la fonction générée :

```
match compare12 n m with
  | Inf → ...
  | Sup → ...
  | Eq → ...
```

Les Reltree sont intéressants, d'une part parce qu'ils sont assez proches de la relation inductive et d'autre part parce qu'ils sont similaires à un langage fonctionnel du point de vue de l'exécution. Ils définissent une interface idéale entre deux étapes de l'extraction de fonctions à partir de relations inductives. Ils permettent d'identifier clairement les mécanismes d'ordonnancement et de fusion de prémisses et de conclusions des constructeurs des relations inductives.

5.5 Schémas de compilation pour les Reltree

Avant de décrire plus formellement les Reltree dans la section suivante, nous commençons par situer leur utilisation dans le processus d'extraction, afin de pouvoir décrire dans quelle phase de la compilation se situe chacun des processus expliqués par la suite. Les schémas de compilation ont été conçus afin que les trois implantations réalisées pendant cette thèse (extraction vers OCaml (ou Haskell), Coq et Focalize) partagent le plus de code possible. Les deux premières sont implantées dans un plugin pour Coq et la dernière dans un plugin intégré au compilateur Focalize. Il y a donc trois schémas de compilation, qui partagent plusieurs étapes de compilation. Pour rendre cela possible, des interfaces précises ont été identifiées. Elles consistent d'une part en une forme normalisée des relations inductives (décrite dans la section 5.1) et d'autre part en un langage cible commun à toutes les implantations que nous nommons MFL (pour "Minimal Functional Language").

La figure 5.6 expose le schéma de compilation général. Ce schéma est étendu pour chaque implantation de l'extraction : l'extraction vers OCaml et Haskell, l'extraction vers Coq et l'extraction vers Focalize. La figure 5.7 présente par exemple le schéma de compilation complet vers OCaml. Dans ce cas, les inductifs de Coq sont normalisés, puis le langage MFL est traduit vers MiniML qui sera lui même traduit vers OCaml ou Haskell par l'extraction native de Coq. L'extraction native réalise les étapes C1, C2 et C3, l'étape C1 consistant à réunir les dépendances des fonctions extraites et à les traduire vers MiniML.

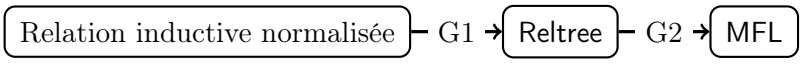


FIGURE 5.6 – Schéma de compilation général

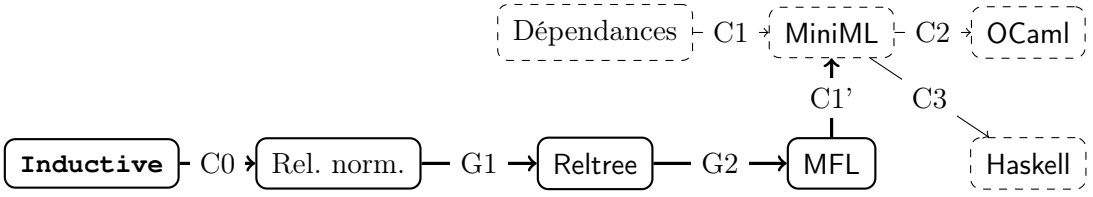


FIGURE 5.7 – Schéma de compilation vers OCaml et Haskell

La forme normalisée des relations inductives est une représentation abstraite de la

définition qui est donnée dans la section 5.1. Même si cette définition a été établie à partir des inductifs de Coq, elle est suffisamment générale pour être compatible avec d'autres outils de preuve. En effet, les contraintes que nous avons ajoutées sur les inductifs permettent de les voir comme des définitions de relations inductives qui sont caractérisées par un type (le type de leurs arguments) et des propriétés sur ces relations. Ainsi, même si les inductifs n'existent pas dans Focalize, on pourra utiliser d'autres constructions de ce langage pour définir des relations inductives.

Le langage fonctionnel MFL est élémentaire. Il s'agit d'un langage ML avec des fonctions récursives, des types inductifs, une construction de filtrage et les tuples. Il n'y a plus de gardes dans les motifs de filtrage, contrairement au langage cible de la première approche. La sémantique de ce langage est formalisée dans la section qui concerne la génération de code (section 5.8).

5.6 Propriétés et invariants des Reltree

Le fait de représenter les relations inductives sous forme d'arbres en vue d'en extraire du code est une idée nouvelle. Cependant, elle n'est pas très éloignée de techniques existantes car nous utilisons le mécanisme de "retour arrière" pour construire les arbres, qui lui est utilisé très largement dans les outils qui génèrent du code à partir de définitions sémantiques [Pettersson 1996]. La principale différence est que ces outils utilisent le "retour arrière" à chaque exécution de la fonction alors que nous ne l'utilisons qu'une fois au moment de la construction du Reltree. Les Reltree servent en quelque sorte à sauvegarder le chemin trouvé avec le mécanisme de "retour arrière". Le déterminisme est essentiel, car il permet de fixer ces chemins au moment de l'extraction, quels que soient les arguments de la fonction à l'exécution. Les Reltree peuvent également être vus comme des arbres de décision, permettant de choisir un constructeur qui va être utilisé pour calculer la sortie de la fonction extraite pour des arguments donnés.

Après avoir décrit de manière informelle la notion de Reltree dans la section précédente, nous allons maintenant en donner une description plus précise dans la section 5.6.1. Pour qu'ils puissent être exploités, les Reltree doivent vérifier certains invariants qui sont décrits

au travers de trois propriétés dans la section 5.6.2.

5.6.1 Définition des Reltree

Définition 2 (Reltree) *Soit d une relation inductive d'arité p. Un Reltree pour d est une forêt dont les arbres sont étiquetés par des termes, et dont les racines et les feuilles ont une étiquette de la forme $d\ t_1\ \ldots\ t_p$. On utilisera les notations suivantes, tirées de la définition des relations inductives :* $\mathsf{Reltree}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1),\ \ldots,\ (d\ t_{k1}\ \ldots\ t_{kp}, \mathsf{Nodes}_k)\})$ *où* Nodes_i *est soit* $\{(T_{i1}, \mathsf{Nodes}_{i1}),\ \ldots,\ (T_{ik}, \mathsf{Nodes}_{ik})\}$ *soit* $d\ t_{i1}\ \ldots\ t_{ip}$.

Les tirets qui masquaient certains termes sur la représentation graphique de la figure 5.5 n'apparaissent pas ici car il s'agissait uniquement d'améliorer la lisibilité pour le lecteur. Dans les algorithmes et les propriétés, on distinguera les différents types de nœud en fonction de leur place dans l'arbre.

5.6.2 Propriétés de bonne formation des Reltree

Afin que l'on puisse générer du code exécutable à partir des Reltree, ces derniers doivent vérifier certains invariants et certaines propriétés par rapport à la définition de la relation inductive. La tâche principale de l'extraction sera alors de produire un Reltree qui vérifie les propriétés décrites dans cette section (la génération de code MFL étant quasiment immédiate une fois le Reltree trouvé). Afin de décrire ces propriétés, nous définissons la fonction treepaths, qui renvoie l'ensemble des chemins d'un Reltree. Par exemple, si search_tree est le Reltree présenté dans la figure 5.5, nous avons :

treepaths(search_tree) = {[search Empty n _; search _ _ NotFound],
[search (Node t1 m t2) n _; compare n m Inf; search t1 n b;
search _ _ Left b], ...}

La première propriété décrit le lien qui existe entre un Reltree pour d et la définition de la relation inductive d. Elle énonce que chaque chemin qui existe dans le Reltree correspond à un constructeur de la relation inductive. Il doit y avoir exactement autant de chemins dans le Reltree que de constructeurs dans la relation inductive. De plus, chaque chemin

doit contenir chaque prémisse du constructeur (à renommage près) et ne les contenir qu'un seule fois. Quant à la conclusion, elle doit apparaître deux fois, une fois au début du chemin ainsi qu'une fois à la fin. Des renommages de variables peuvent apparaître dans les Reltree. Cela se produit lorsqu'il y a fusion de deux constructeurs qui ne sont pas syntaxiquement identiques. Lorsqu'un nœud subit un renommage, tous ses enfants le subissent également.

Propriété 1 (Conformité avec la spécification)

Soit un Reltree r pour d. r est conforme à la relation inductive d et au mode m si et seulement si $\mathrm{SC}(r)$.

Avec

$$\mathrm{SC}(r) \triangleq \begin{cases} \mathrm{SC_A}(r) \wedge \mathrm{SC_B}(r) \\ \textit{où } \mathrm{SC_A}(r) \triangleq \forall C \in \Gamma, \exists! p \in \mathsf{treepaths}(r), \mathrm{SC}'(p, C) \\ \textit{et } \mathrm{SC_B}(r) \triangleq \forall p \in \mathsf{treepaths}(r), \exists! C \in \Gamma, \mathrm{SC}'(p, C) \end{cases}$$

où SC' *est la conformité d'un chemin* $[T_0;\ T_1;\ \ldots;\ T_n;\ T_{n+1}]$ *pour un constructeur donné nommé* C *définie comme suit :*

$$\mathrm{SC}'([T_0;\ T_1;\ \ldots;\ T_n;\ T_{n+1}], C) \triangleq$$
$$\begin{cases} \exists \sigma, \exists \pi, n = Card(P(C)) \wedge \forall i \in 1..n, T_i = \sigma(P(C, \pi(i))) \wedge \\ \mathsf{out}(T_{n+1}, m) = \mathsf{out}(\sigma(\mathsf{concl}(\Gamma(C))), m) \wedge \\ \quad \mathsf{in}(T_0, m) = \mathsf{in}(\sigma(\mathsf{concl}(\Gamma(C)), m)) \end{cases}$$

où σ *est un renommage de variables,* π *une permutation de* $1..n$ *et* concl *une fonction qui renvoie la conclusion d'un constructeur.*

Les deux propriétés restantes sont des invariants sur les Reltree qui assurent qu'il sera possible de générer du code fonctionnel bien typé à partir d'un Reltree lors de l'étape de génération de code. L'analyse de cohérence de mode est similaire à celle présentée dans la section 5.3.1, mais elle est réalisée sur les arbres au lieu des constructeurs la relation inductive. L'ensemble des variables connues est ici appelé S, et on commence par y ajouter les entrées des conclusions. Puis on teste pour chaque prémisse que les variables de ses entrées sont connues avant d'ajouter dans S les variables de ses sorties.

Propriété 2 (Analyse de cohérence de mode)

Soit un Reltree r pour une relation inductive d. Un mode m est dit cohérent pour r si et seulement si $\mathrm{MCA}(r)$.

Avec

$$\mathrm{MCA}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1);\ \ldots;\ (d\ t_{n1}\ \ldots\ t_{np}, \mathsf{Nodes}_n)\}) \triangleq$$
$$\forall i, i \in 1..n, \mathrm{MCA}'(\mathsf{Nodes}_i, \mathsf{invars}(d\ t_{i1}\ \ldots\ t_{ip}, m))$$

où MCA′ *est définie comme suit :*

$$\mathrm{MCA}'(nodes, S) \triangleq \begin{cases} \textit{si } nodes \textit{ est } d\ t_1\ \ldots\ t_p \textit{ alors } \mathsf{outvars}(d\ t_1\ \ldots\ t_p, m) \subseteq S \\ \textit{si } nodes \textit{ est } \{(T_1, \mathsf{Nodes}_1);\ \ldots;\ (T_n, \mathsf{Nodes}_n)\} \textit{ alors} \\ \quad \forall i, i \in 1..n, \mathsf{invars}(T_i, \mathcal{M}(T_i)) \subseteq S \wedge \\ \quad \mathrm{MCA}'(\mathsf{Nodes}_i, S \cup \mathsf{outvars}(T_i, \mathcal{M}(T_i))) \end{cases}$$

La troisième propriété assure qu'il sera possible de former des filtrages valides à partir d'un Reltree, avec des motifs de filtrage qui ne se recouvrent pas. Peu importe le langage cible final, nous considérons ici le langage MFL (décrit dans la section 5.8) avec un filtrage à la ML et la possibilité de construire des tuples. Nous vérifions le non recouvrement des motifs de filtrage mais pas la complétude du filtrage. Les motifs d'un même filtrage seront les sorties des nœuds contenant des prémisses et qui partagent le même nœud père. La propriété 3 vérifie également que les entrées des nœuds qui partagent un même père sont identiques, car ces termes seront fusionnés dans la fonction générée.

Propriété 3 (Non recouvrement)

Soit un Reltree r pour d. r est non recouvrant avec le mode m si et seulement si $\mathsf{NO}(r)$.

Avec

$$\mathsf{NO}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1);\ \ldots;\ (d\ t_{n1}\ \ldots\ t_{np}, \mathsf{Nodes}_n)\}) \triangleq$$
$$\forall i, i \in 1..n, \mathsf{NO}'(\mathsf{Nodes}_i) \wedge \forall j, j \in 1..n, j \neq i \Rightarrow$$
$$\mathsf{in}(d\ t_{i1}\ \ldots\ t_{ip}, m)\ \textit{et}\ \mathsf{in}(d\ t_{j1}\ \ldots\ t_{jp}, m)\ \textit{ne sont pas unifiables}$$

où la propriété NO′ *est définie comme suit :*

$$\mathsf{NO}'(nodes) \triangleq$$
$$\begin{cases} \textit{si } nodes \textit{ est } d\ t_1\ \ldots\ t_p \textit{ alors } \mathsf{true} \\ \textit{si } nodes \textit{ est } \{(T_1, \mathsf{Nodes}_1);\ \ldots;\ (T_k, \mathsf{Nodes}_k)\} \textit{ alors} \\ \quad \forall i, i \in 1..k, \begin{cases} \mathsf{NO}'(\mathsf{Nodes}_i) \wedge \\ \quad \forall j, j \in 1..k, j \neq i \Rightarrow \mathsf{in}(T_i, \mathcal{M}(T_i)) = \mathsf{in}(T_j, \mathcal{M}(T_j)) \wedge \\ \quad \mathsf{out}(T_i, \mathcal{M}(T_i))\ \textit{et}\ \mathsf{out}(T_j, \mathcal{M}(T_j))\ \textit{ne sont pas unifiables} \end{cases} \end{cases}$$

Les filtrages dans la fonction générée auront des motifs non recouvrants si le Reltree qui sert de support à la génération de la fonction est non recouvrant.

On dira qu'un Reltree est valide pour d (avec le mode m) lorsqu'il vérifie les propriétés 1, 2 et 3. On définit la propriété 4 :

Propriété 4 (Validité)
Soit un Reltree r pour d. r est un Reltree valide pour d avec le mode m si et seulement si $\mathsf{VAL}(r)$.
Avec
$\mathsf{VAL}(r) \triangleq \mathsf{SC}(r) \wedge \mathsf{MCA}(r) \wedge \mathsf{NO}(r)$

5.7 Construction des arbres

Cette section formalise l'algorithme de construction d'un Reltree pour une relation inductive dont on veut extraire une fonction avec un mode donné. Cet algorithme ne tient pas compte des extensions qui sont décrites dans la section 5.9. L'algorithme étant complexe, nous commençons par décrire le principe général dans la section 5.7.2, après avoir introduit quelques notations dans la section 5.7.1. Puis nous poursuivons avec la structure de l'algorithme dans la section 5.7.3 et l'algorithme proprement dit dans la section 5.7.4.

5.7.1 Rappels et notations

Nous commençons par rassembler dans cette section l'ensemble des notions et des notations nécessaires à la formalisation de l'algorithme de construction des Reltree à partir d'une relation inductive. Nous listons pour cela les définitions et notations utiles appartenant aux sections précédentes et nous y ajoutons quelques nouvelles définitions.

Relations inductives Les Reltree sont construits directement à partir des relations inductives dont la définition est donnée dans la section 5.1. Nous conserverons les notations utilisées dans cette définition. Rappelons que d désigne le nom de la relation inductive dont on considère l'extraction alors que d_i est le nom d'une relation inductive utilisée dans la définition de d. Nous utiliserons également le raccourci suivants : C ou C_i pour

un constructeur. Dans la définition de la section 5.1, les constructeurs sont nommés dans l'environnement Γ. Lorsque l'on utilise la notation C_i, on ne tient plus compte de ces noms.

Outre les fonctions in, invars, out, outvars et rel définies précédemment, on définit les fonctions prems et concl qui renvoient respectivement l'ensemble des prémisses et la conclusion d'un constructeur, afin de manipuler les relations inductives et leur définition.

Modes d'extraction L'environnement $\mathcal{M}$ contient les modes d'extraction des différentes relations inductives. Puisque l'on ne considère ici qu'un mode d'extraction pour chaque relation inductive, on peut alléger les notations, en notant m le mode d'extraction de d ($\mathcal{M}(d)$) et m_i ($\mathcal{M}(d_i)$) le mode d'extraction de la relation d_i.

Notations et fonctions pour les Reltree La définition des Reltree est donnée dans la définition 2. On ajoute les notations suivantes :

— Les nœuds sont notés n, avec ou sans indice. On introduit le type *node* comme étant l'ensemble des nœuds possibles (sans les feuilles), qu'ils soient de la forme $(d\ t_1\ \ldots\ t_p, \mathsf{Nodes})$ qui est un nœud issu d'une conclusion ou bien de la forme (T, Nodes) qui est un nœud issu d'une prémisse.

— Les feuilles sont notées l, on introduit de la même manière le type $leaf$ comme étant l'ensemble des feuilles qu'il est possible d'écrire.

On définit également la fonction children qui renvoie l'ensemble des fils d'un nœud et la fonction label qui prend un nœud en argument et qui renvoie son étiquette, c'est à dire un terme.

5.7.2 Principe de l'algorithme

Les Reltree sont construits en partant d'un Reltree vide dans lequel sont ajoutées une à une les branches pour chaque constructeur présent dans la relation inductive. Un Reltree dont la construction est terminée possède autant de chemins qu'il y a de constructeurs dans la relation d. Chaque ajout des nœuds qui correspondent à un constructeur constitue une étape de l'algorithme et lors de chacune de ces étapes, des choix interviennent. Si les propriétés que l'on a définies dans la section 5.6 imposent que chaque chemin dans l'arbre

commence et se termine par la conclusion d'un constructeur et qu'entre ces deux nœuds on trouve l'ensemble des prémisses, il reste alors à déterminer l'ordonnancement des prémisses, et le fait de fusionner ou non certains nœuds.

Parmi ces choix, certains doivent être réalisés arbitrairement car on se sait pas à l'avance quel choix sera bon ou mauvais. Cela implique généralement la mise en place d'un mécanisme de "retour arrière", mais il est également possible de calculer à chaque étape un ensemble de Reltree valides (pour d), qui résultent des différents choix valides. Il faut alors réaliser les étapes suivantes de l'algorithme sur chacun des Reltree obtenus à l'étape précédente. Le mécanisme de "retour arrière" requiert de se souvenir de tous les choix qui ont été faits et des choix qui restaient à explorer, ce qui peut conduire dans certains cas à une explosion de la mémoire nécessaire pour exécuter l'algorithme. En revanche, si l'on calcule tous les choix valides, il n'est pas nécessaire de conserver tous les Reltree calculés car certains d'entre eux sont équivalents. Pour ces raisons, j'ai choisi cette dernière méthode, présentée ci-dessous.

5.7.3 Décomposition de l'algorithme

Afin de clarifier la formalisation de l'algorithme, qui contient plusieurs étapes mutuellement récursives, nous commençons par donner la signature des sous-algorithmes qui composent l'algorithme principal dans le tableau 5.1.

L'algorithme principal est noté $\mathcal{R}$. Il renvoie un ensemble de Reltree valides à partir d'une relation inductive nommée d ainsi que du mode d'extraction de cette relation et de toutes les autres relations inductives utilisées pour définir d. Les modes d'extraction n'apparaissent pas dans la signature des algorithmes car on dispose par ailleurs de $\mathcal{M}$ qui associe à chaque relation inductive son mode d'extraction.

L'algorithme $\mathcal{R}_{constrs}$ construit un ensemble de Reltree à partir de l'ensemble des constructeurs.

L'algorithme $\mathcal{S}$ réalise une étape de l'algorithme : il calcule tous les Reltree valides qui peuvent résulter de l'insertion d'un constructeur C dans un ensemble de Reltree $trees$.

L'algorithme $\mathcal{C}$ est similaire à $\mathcal{S}$, mais il ne calcule les résultats que pour un seul Reltree

$\mathcal{R}$	rel **: InductiveRelation** $\rightarrow$ **Reltree Set**
$\mathcal{R}_{contrs}$	Γ **: constructor Set** $\rightarrow$ **Reltree Set** où Γ est un ensemble de constructeurs,
$\mathcal{S}$	ρ **: Reltree Set** $\rightarrow$ C **: constructor** $\rightarrow$ **Reltree Set** où ρ est un ensemble de Reltree et C un constructeur,
$\mathcal{C}$	r **: Reltree** $\rightarrow$ C **: constructor** $\rightarrow$ **Reltree Set**
$\mathcal{C}_{init}$	r **: Reltree** $\rightarrow$ τ **: term Set** $\rightarrow$ t **: term** $\rightarrow$ **Reltree Set** où r est un Reltree, τ un ensemble de prémisses et t la conclusion du constructeur,
$\mathcal{C}_{prems}$	η **: node Set** $\rightarrow$ τ **: term Set** $\rightarrow$ t **: term** $\rightarrow$ kv **: var Set** $\rightarrow$ **(node Set) Set** où η est un ensemble de nœuds qui partagent le même père, τ un ensemble de prémisses, t la conclusion du constructeur et kv l'ensemble des variables connues,
$\mathcal{C}_{prem}$	η **: node Set** $\rightarrow$ T **: term** $\rightarrow$ τ **: term Set** $\rightarrow$ t **: term** $\rightarrow$ kv **: var Set** $\rightarrow$ **(node Set) Set** où η est un ensemble de nœuds qui partagent le même père, T la prémisse à insérer, τ les prémisses restantes, t la conclusion du constructeur et kv l'ensemble des variables connues,
$\mathcal{C}_{concl}$	t **: term** $\rightarrow$ kv **: var Set** $\rightarrow$ **leaf Set** où t est la conclusion du constructeur et kv l'ensemble des variables connues.

où InductiveRelation, Reltree, constructor et term sont respectivement les types des relations inductives, des Reltree, des constructeurs des relations inductives et des termes tels qu'ils apparaissent dans la définition des relations inductives donnée dans la section 5.1. Le type var est celui des variables.

TABLE 5.1 – Étapes de l'algorithme $\mathcal{R}$

rt.

Les algorithmes $\mathcal{C}_{init}$, $\mathcal{C}_{prems}$, $\mathcal{C}_{prem}$ et $\mathcal{C}_{concl}$ sont des étapes de l'algorithme $\mathcal{C}$. $\mathcal{C}_{init}$ construit le nœud racine à partir de la *conclusion* et l'insère dans le Reltree r, $\mathcal{C}_{prems}$ insère les prémisses τ une par une dans le sous-Reltree η en faisant appel à $\mathcal{C}_{prem}$ pour chaque prémisse (T), et $\mathcal{C}_{concl}$ insère la feuille dans l'arbre à partir de la conclusion t. L'ensemble de variables kv contient les variables connues, qui sert à la vérification de la propriété 2.

5.7.4 Algorithme de construction d'un ensemble de Reltree

La description suivante des différentes étapes de l'algorithme est donnée en pseudo code. Elle doit permettre d'implanter l'algorithme dans un système quelconque. Afin de montrer comment l'algorithme fonctionne, un exemple d'arbre est construit dans la suite de cette section en utilisant cet algorithme. Le mode m avec lequel on extrait une fonction à partir de la relation inductive d, ainsi que d et $\mathcal{M}$ (l'ensemble des modes d'extraction) sont des paramètres implicites de tous les algorithmes. De plus, σ désigne un renommage de variables.

5.7.4.1 Algorithme en pseudo code

$\boldsymbol{\mathcal{C}_{concl}(t,\ kv)}\ \triangleq$

▷ SI $\mathsf{outvars}(t, m) \subseteq kv$ ALORS

— Soit $d\ t_1\ \ldots\ t_p = t$.

— RENVOYER $\{(d\ t_1\ \ldots\ t_p)\}$

▷ SINON RENVOYER $\emptyset$

L'algorithme $\mathcal{C}_{concl}$ crée un nœud correspondant à la conclusion d'un constructeur.

$\boldsymbol{\mathcal{C}_{prem}(\eta, T, \tau, t,\ kv)}\ \triangleq$

▷ SI $\tau \neq \emptyset \wedge \exists! n_0 \in \eta, \exists \sigma, \mathsf{label}(n_0) = \sigma(T)$ ALORS

— Soit $t_0 = \mathsf{label}(n_0)$.

— On définit :

$R = \mathcal{C}_{prems}(\mathsf{children}(n_0),\ \{\sigma(T') | T' \in \tau\},\ \sigma(t),\ kv \cup \mathsf{outvars}(\sigma(T), \mathcal{M}(T)))$.

— RENVOYER $\{\{(t_0, r_k)\} \cup (\eta - \{n_0\}) | r_k \in R\}$.

▷ SINON SI $\mathsf{invars}(T, \mathcal{M}(T)) \subseteq kv \wedge \forall n \in \eta,$
$\begin{cases} \mathsf{rel}(\mathsf{label}(n)) = \mathsf{rel}(T) \\ \mathsf{in}(\mathsf{label}(n), \mathcal{M}(n)) = \mathsf{in}(T, \mathcal{M}(T)) \\ \mathsf{out}(\mathsf{label}(n), \mathcal{M}(n)) \text{ et } \mathsf{out}(T, \mathcal{M}(T)) \text{ non unifiables} \end{cases}$ ALORS

— On définit $R = \mathcal{C}_{prems}(\emptyset,\ \tau,\ t,\ kv \cup \mathsf{outvars}(T))$.

— RENVOYER $\{\{(T, r_k)\} \cup \eta | r_k \in R\}$.

▷ SINON RENVOYER $\emptyset$

Où σ est un renommage de variables qui ne renomme aucune variable de l'ensemble kv.

L'algorithme $\mathcal{C}_{prem}$ distingue trois cas :

— dans le premier cas, la prémisse T est fusionnée avec un nœud existant du Reltree (si le nœud n_0 existe, il est nécessairement unique d'après la propriété 3),

— dans le second cas, la prémisse T est insérée dans un nouveau nœud, lui-même ajouté à l'ensemble η,

— et dans le dernier cas, $\mathcal{C}_{prem}$ renvoie un ensemble vide de Reltree.

$\boldsymbol{\mathcal{C}_{prems}(\eta, \tau, t,\ kv)} \triangleq$

▷ SI $\tau = \emptyset$ ALORS RENVOYER $\mathcal{C}_{concl}(t,\ kv)$

▷ SINON RENVOYER $\bigcup_{T \in \tau}(\mathcal{C}_{prem}(\eta, T, \tau - \{T\}, t,\ kv))$

L'algorithme $\mathcal{C}_{prems}$ permet de tester toutes les permutations des prémisses de l'ensemble τ, et calcule l'union des solutions obtenues pour chaque permutation.

Dans les algorithmes suivants, Nodes_i désigne un ensemble de nœuds. Cette notation provient de la définition des Reltree.

$\boldsymbol{\mathcal{C}_{init}(\mathbf{Reltree}(\eta), \tau, d\ t_1\ \ldots\ t_p)} \triangleq$

▷ SI $\exists! n_0 \in \eta, \exists \sigma, \exists t_{01} \ldots t_{0p},$
$\exists \mathsf{Nodes}_0, \begin{cases} n_0 = (d\ t_{01}\ \ldots\ t_{0p}, \mathsf{Nodes}_0) \\ \mathsf{in}(d\ t_{01}\ \ldots\ t_{0p}, m) = \mathsf{in}(\sigma(d\ t_1\ \ldots\ t_p), m) \end{cases}$ ALORS

— On définit :

$R = \mathcal{C}_{prems}(\mathsf{Nodes}_0,\ \{\sigma(T) | T \in \tau\},\ \sigma(d\ t_1\ \ldots\ t_p),\ \mathsf{invars}(\sigma(d\ t_1\ \ldots\ t_p), m))$.

— RENVOYER $\{\mathsf{Reltree}(\{(d\ t_{01}\ \ldots\ t_{0p}, r_k)\} \cup (\eta - \{n_0\})) | r_k \in R\}$.

▷ SINON SI $\forall n_i \in \eta, \exists t_{i1}\ \ldots\ t_{ip}, \exists \mathsf{Nodes}_i,$
$\begin{cases} n_i = (d\ t_{i1}\ \ldots\ t_{ip}, \mathsf{Nodes}_i) \\ \mathsf{in}(d\ t_{i1}\ \ldots\ t_{ip}, m) \text{ et } \mathsf{in}(d\ t_1\ \ldots\ t_p, m) \text{ non unifiables} \end{cases}$ ALORS

— On définit $R = \mathcal{C}_{prems}(\emptyset,\ \tau,\ d\ t_1\ \ldots\ t_p,\ \mathsf{invars}(d\ t_1\ \ldots\ t_p, m))$.

— RENVOYER $\{\mathsf{Reltree}(\eta \cup \{(d\ t_1\ \ldots\ t_p, r_k)\})|r_k \in R\}$.

▷ SINON RENVOYER $\emptyset$

Où σ est un renommage de variables qui ne renomme aucune variable de l'ensemble kv.

$\mathcal{C}_{init}$ distingue trois cas :

— dans le premier cas, la conclusion $d\ t_1\ \ldots\ t_p$ est fusionnée avec un nœud existant du Reltree (si le nœud n_0 existe, il est nécessairement unique d'après la propriété 3),

— dans le second cas, la conclusion $d\ t_1\ \ldots\ t_p$ est insérée dans un nouveau nœud, lui-même ajouté à l'ensemble η,

— et dans le dernier cas, $\mathcal{C}_{init}$ renvoie un ensemble vide de Reltree.

$\mathcal{C}(r, C)$ $\triangleq$

▷ Soit C de la forme :

$\forall x_1 : X_1, \ldots x_i : X_i, \ldots x_n : X_n . T_1 \to \ldots T_j \to \ldots T_m \to (d\ t_1\ \ldots\ t_p)$.

$\mathcal{C}_{init}(r, \{T_j | j \in 1..n\}, d\ t_1\ \ldots\ t_p)$

L'algorithme $\mathcal{C}$ permet de déstructurer le constructeur C de la relation inductive, puis fait appel à $\mathcal{C}_{init}$.

$\mathcal{S}(\rho, C)$ $\triangleq$ $\bigcup_{r \in \rho} \mathcal{C}(r, C)$

L'algorithme $\mathcal{S}$ calcule toutes les possibilités d'insertion du constructeur C dans chaque Reltree "partiel" de l'ensemble ρ. Un Reltree partiel est un Reltree dans lequel tous les constructeurs ne sont pas encore présents.

$\mathcal{R}_{constrs}(\Gamma)$ $\triangleq$

▷ SI $\Gamma = \emptyset$ ALORS RENVOYER $\{\mathsf{Reltree}(\emptyset)\}$

▷ SINON

— Soit $C \in \Gamma$.

— On définit $R = R_{constrs}(\Gamma - \{C\})$.

— RENVOYER $\mathcal{S}(R, C)$.

L'algorithme $\mathcal{R}_{constrs}$ calcule l'ensemble des Reltree valides que l'on peut construire à partir des constructeurs contenus dans Γ.

$\boldsymbol{\mathcal{R}(\mathsf{Ind}(d : \tau, \Gamma))}$ $\triangleq$ RENVOYER $\mathcal{R}_{constrs}(\Gamma)$

L'algorithme $\mathcal{R}$ déstructure la relation inductive et fait appel à $\mathcal{R}_{constrs}$.

5.7.4.2 Exemple

Nous reprenons ici l'exemple de la sémantique du langage IMP. En particulier nous allons appliquer l'algorithme $\mathcal{R}$ à la relation `exec` de la figure 4.1. Pour des raisons de place, nous ne déroulons pas intégralement l'algorithme ; de plus, certaines étapes seront résumées.

Tout d'abord, rappelons que l'algorithme de construction traite les constructeurs de la relation inductive un par un, et que nous conservons, à chaque étape non pas un seul, mais un ensemble de Reltree. Chaque étape consiste donc à calculer, pour un ensemble de Reltree ρ, un nouvel ensemble de Reltree ρ' qui regroupe toutes les possibilités d'insertion d'un constructeur C dans chacun des Reltree appartenant à ρ.

L'algorithme $\mathcal{R}$ ne sert qu'à déstructurer la relation inductive pour en extraire les constructeurs. L'algorithme $\mathcal{R}_{constrs}$ sélectionne les constructeurs un à un et pour chacun d'entre eux, appelle l'algorithme $\mathcal{S}$ qui réalise une étape de l'algorithme. Avec les notations du paragraphe précédent, on peut écrire : $\rho' = S(\rho, C)$.

L'ordre d'insertion des constructeurs n'ayant pas d'importance, nous fixons ici l'ordre suivant :

`exec_while2` ; `exec_while1` ; `exec_test1` ; `exec_test2` ; `exec_seq` ; `exec_affect` ; `exec_skip`.

Concernant le premier constructeur, l'appel à $\mathcal{S}$ est le suivant :

$\mathcal{S}(\emptyset, \forall b\ c\ s\ s'\ s'',\ evalbool\ b\ s\ true\ \rightarrow\ exec\ c\ s\ s''\ \rightarrow\ exec\ (while\ b\ c)\ s''\ s'\ \rightarrow$
$exec\ (while\ b\ c)\ s\ s')$.

Ensuite, l'algorithme $\mathcal{C}$ déstructure le constructeur en deux entités : ensemble de prémisses et conclusion. $\mathcal{C}_{init}$ insère la conclusion $(exec\ (while\ b\ c)\ s\ s')$ dans un nouveau nœud ou la fusionne avec un nœud existant. Pour l'instant, le Reltree est vide, donc un nouveau nœud est créé et l'algorithme se poursuit avec l'appel à $\mathcal{C}_{prems}$. Le rôle de $\mathcal{C}_{prems}$, couplé à $\mathcal{C}_{prem}$, est d'insérer les prémisses dans le Reltree en testant toutes les permutations possibles. Les seules permutations qui sont conservées sont celles qui vérifient certaines propriétés. Essentiellement, $\mathcal{C}_{prem}$ assure que pour toute prémisse T, $\mathsf{invars}(T) \subseteq kv$ où kv est l'ensemble des variables connues issues des nœuds précédents. Dans le cas du constructeur `exec_while2`, les permutations suivantes sont conservées :

1. `evalbool b s true ; exec c s s'' ; exec (while b c) s'' s'`
2. `exec c s s'' ; evalbool b s true ; exec (while b c) s'' s'`
3. `exec c s s'' ; exec (while b c) s'' s' ; evalbool b s true`

Enfin, l'algorithme $\mathcal{C}_{concl}$ insère la conclusion. Toutes les variables utilisées dans la conclusion ($\mathsf{outvars}(exec\ (while\ b\ c)\ s\ s', m) = \{s'\}$) sont bien connues ($\{s'\} \subseteq kv$). On obtient alors, après une étape de l'algorithme, les trois Reltree de la figure 5.8. Soit S_1 l'ensemble de ces trois Reltree.

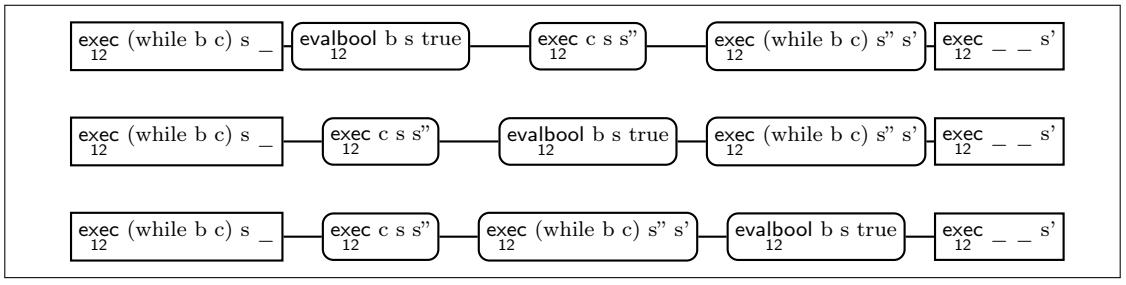


FIGURE 5.8 – Ensemble de Reltree obtenu après la premère étape de construction (S_1)

De la même façon, l'étape suivante se charge d'insérer le constructeur `exec_while1` dans Reltree S_1. Le deuxième appel à $\mathcal{S}$ est le suivant :
$\mathcal{S}(S_1, \forall b\ c\ s,\ evalbool\ b\ s\ false\ \rightarrow\ exec\ (while\ b\ c)\ s\ s)$. L'algorithme $\mathcal{S}$ calcule toutes les possibilités pour insérer le constructeur `exec_while1` dans chacun des trois Reltree de S_1.

— Pour le premier Reltree rt_1, $\mathcal{C}_{init}$ fusionne la conclusion de `exec_while1` avec le nœud racine de rt_1, car les entrées des conclusions des constructeurs `exec_while1`

et `exec_while2` sont identiques. Ensuite, $\mathcal{C}_{prem}$ insère la prémisse de `exec_while1` dans le sous-arbre dont la racine est étiquetée : *evalbool b s true*. Deux options sont possibles dans l'algorithme $\mathcal{C}_{prem}$: fusionner la prémisse (ici *evalbool b s false*) dans un nœud existant ou bien créer un nouveau nœud. Ici, les sorties des prémisses (*true* et *false*) ne sont pas unifiables, c'est un nouveau nœud qui est créé. Les variables en sortie de la conclusion ($\mathsf{outvars}(exec\ (while\ b\ c)\ s\ s, m) = \{s\}$) étant connues, $\mathcal{C}_{concl}$ insère le nœud qui correspond à la conclusion. Le résultat est constitué d'un unique Reltree qui est présenté dans la figure 5.9.

— Pour le second Reltree rt_2, la conclusion peut être fusionnée comme dans le cas de rt_1. En revanche, l'insertion de la prémisse *evalbool b s false* par l'algorithme $\mathcal{C}_{prem}$ n'est pas possible car $\mathsf{rel}(evalbool\ b\ s\ false) = evalbool$ est différent de $\mathsf{rel}(exec\ c\ s\ s'') = exec$. Le résultat de l'insertion dans rt_2 est donc l'ensemble vide.

— Pour le dernier Reltree rt_3, le résultat est le même que pour rt_2 car le début des arbres est identique.

Finalement, l'algorithme $\mathcal{S}$ calcule l'union des résultats précédents et on obtient un seul singleton contenant le Reltree de la figure 5.9.

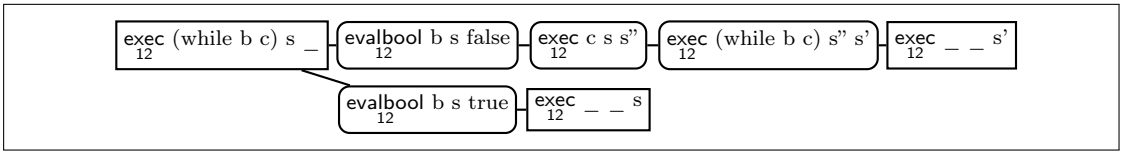


FIGURE 5.9 – Reltree résultant de l'insertion de $exec\_while1$ dans rt_1

Les autres constructeurs sont insérés de façon similaire. Au final, on obtient un seul Reltree, celui de la figure 5.10. Ce dernier peut être utilisé pour générer un programme fonctionnel, car il vérifie les propriétés 1, 2 et 3 de construction des Reltree.

5.7.5 Propriétés de l'algorithme de construction des Reltree

Nous décrivons dans le tableau 5.2 les propriétés qui caractérisent les arguments, appelées préconditions (colonne de gauche), ainsi que les propriétés attendues des valeurs renvoyées, appelées postconditions (colonne de droite), et ce pour toutes les fonctions du tableau 5.1. Ces propriétés sont relatives aux propriétés 2 et 3 des Reltree.

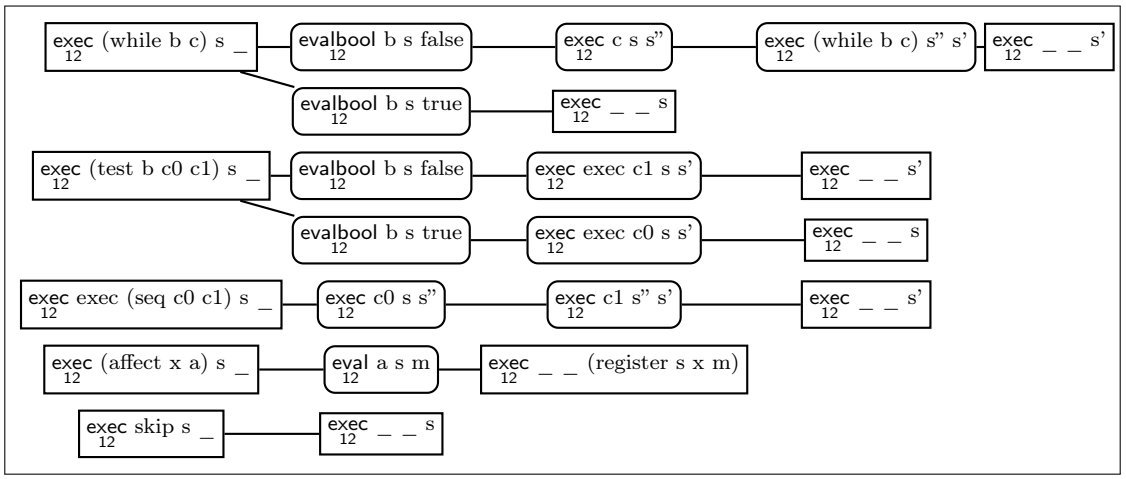


FIGURE 5.10 – Reltree pour la relation inductive `exec`

Algorithme	Précondition	Postcondition
$\mathcal{R}$		$\forall rt \in \$, \mathrm{MCA}(rt) \wedge \mathrm{NO}(rt)$
$\mathcal{R}_{constrs}$		$\forall rt \in \$, \mathrm{MCA}(rt) \wedge \mathrm{NO}(rt)$
$\mathcal{S}$	$\forall rt \in \rho, \mathrm{MCA}(rt) \wedge \mathrm{NO}(rt)$	$\forall rt \in \$, \mathrm{MCA}(rt) \wedge \mathrm{NO}(rt)$
$\mathcal{C}$	$\mathrm{MCA}(rt) \wedge \mathrm{NO}(rt)$	$\forall rt \in \$, \mathrm{MCA}(rt) \wedge \mathrm{NO}(rt)$
$\mathcal{C}_{init}$	$\mathrm{MCA}(rt) \wedge \mathrm{NO}(rt)$	$\forall rt \in \$, \mathrm{MCA}(rt) \wedge \mathrm{NO}(rt)$
$\mathcal{C}_{prems}$	$\mathrm{MCA}'(\eta, kv) \wedge \mathrm{NO}'(\eta)$	$\forall \eta \in \$, \mathrm{MCA}'(\eta, kv) \wedge \mathrm{NO}'(\eta)$
$\mathcal{C}_{prem}$	$\mathrm{MCA}'(\eta, kv) \wedge \mathrm{NO}'(\eta)$	$\forall \eta \in \$, \mathrm{MCA}'(\eta, kv) \wedge \mathrm{NO}'(\eta)$
$\mathcal{C}_{concl}$		$\forall \eta \in \$, \mathrm{MCA}'(\eta, kv) \wedge \mathrm{NO}'(\eta)$

TABLE 5.2 – Contrats des algorithmes de construction des Reltree

Dans les postconditions du tableau 5.2, \$ désigne le résultat renvoyé par la fonction. Ainsi la première postcondition du tableau se lit de la manière suivante : “chacun des Reltree de l’ensemble résultat vérifie les prédicats MCA et NO”.

Nous avons fait le choix de générer la preuve de correction des fonctions extraites à partir de relation inductives. Ce travail est traité dans le chapitre 7. Cependant, nous donnons ici une partie de la preuve de correction partielle de l’algorithme de construction des Reltree. L’objectif est d’augmenter la confiance en cet algorithme compliqué. La preuve est partielle car nous ne prouvons pas ici la propriété 1.

Nous faisons ici une preuve modulaire, dont l’objectif est de montrer que les Reltree construits avec l’algorithme $\mathcal{R}$ vérifient les propriétés 2 et 3. Pour cela, nous montrons que chaque algorithme vérifie la postcondition écrite dans le tableau 5.2, sous condition de sa

précondition. Tout appel à un autre algorithme requerra de montrer que la précondition de ce dernier est satisfaite. On pourra alors utiliser dans la preuve la postcondition associée à cet appel.

5.7.5.1 Preuve de correction partielle des algorithmes $\mathcal{R}$, $\mathcal{R}_{constrs}$, $\mathcal{S}$ et $\mathcal{C}$

La preuve de correction de ces algorithmes par rapport à leur contrat est triviale. En effet, chacun de ces algorithmes $\mathcal{A}$ fait appel à un second algorithme $\mathcal{B}$, dont la précondition est impliquée par celle de $\mathcal{A}$. En supposant la précondition de $\mathcal{A}$, on obtient donc que celle de $\mathcal{B}$ est vérifiée. On peut alors en déduire la postcondition de $\mathcal{B}$, dont on déduit trivialement la postcondition de $\mathcal{A}$.

5.7.5.2 Preuve de correction partielle de $\mathcal{C}_{init}(\mathsf{Reltree}(\eta), \tau, d\ t_1\ \ldots\ t_p)$

Nous considérons la précondition $\mathrm{MCA}(r) \wedge \mathrm{NO}(r)$ (1) avec $r = \mathsf{Reltree}(\eta)$.
Montrons que $\forall r \in \$, \mathrm{MCA}(r) \wedge \mathrm{NO}(r)$.
Pour cela, on distingue 3 cas qui suivent la structure de l'algorithme $\mathcal{C}_{init}$:

1. $\exists! n_0 \in \eta, \exists \sigma, \exists t_{01} \ldots t_{0p}, \exists \mathsf{Nodes}_0,$
   $\begin{cases} n_0 = (d\ t_{01}\ \ldots\ t_{0p}, \mathsf{Nodes}_0) \\ \mathsf{in}(d\ t_{01}\ \ldots\ t_{0p}, m) = \mathsf{in}(\sigma(d\ t_1\ \ldots\ t_p), m) \end{cases}$ (H)
   D'après (1) et la définition de MCA :
   $\forall n \in \eta, \mathrm{MCA}'(\mathsf{children}(n), \mathsf{invars}(\mathsf{label}(n), m))$ (2)
   De même, on a $\forall n \in \eta, \mathrm{NO}'(\mathsf{children}(n))$ (3)
   Soit $R = \mathcal{C}_{prems}(\mathsf{Nodes}_0,\ \{\sigma(T) | T \in \tau\},\ \sigma(d\ t_1\ \ldots\ t_p),\ \mathsf{invars}(\sigma(d\ t_1\ \ldots\ t_p), m))$.
   Vérifions que la précondition pour l'appel à $\mathcal{C}_{prems}$ est vérifiée :
   On applique (2) avec n_0 : $\mathrm{MCA}'(\mathsf{Nodes}_0, \mathsf{invars}(d\ t_{01}\ \ldots\ t_{0p}, m))$
   d'où $\mathrm{MCA}'(\mathsf{Nodes}_0, \mathsf{invars}(\sigma(d\ t_1\ \ldots\ t_p), m))$ (4) d'après la seconde égalité de (H)
   On applique également (3) avec n_0 : $\mathrm{NO}'(\mathsf{Nodes}_0)$ (5)
   (4) et (5) permettent de vérifier la précondition de $\mathcal{C}_{prems}$. D'après la postcondition de $\mathcal{C}_{prems}$, on a :
   $\forall r_k \in R, \mathrm{MCA}'(r_k, \mathsf{invars}(\sigma(d\ t_1\ \ldots\ t_p), m))$ (6)
   et $\forall r_k \in R, \mathrm{NO}'(r_k)$ (7)
   Dans ce cas, le résultat de $\mathcal{C}_{init}$ est $\{\mathsf{Reltree}(\{(d\ t_{01}\ \ldots\ t_{0p}, r_k)\} \cup (\eta - \{n_0\})) | r_k \in R\}$.

Il s'agit donc de montrer que $\forall r_k \in R, \mathrm{MCA}(\mathsf{Reltree}(\{(d\ t_{01}\ \ldots\ t_{0p}, r_k)\} \cup (\eta - \{n_0\})))$ (C_1)

et que $\forall r_k \in R, \mathrm{NO}(\mathsf{Reltree}(\{(d\ t_{01}\ \ldots\ t_{0p}, r_k)\} \cup (\eta - \{n_0\})))$ (C_2).

Pour démontrer (C_1), nous montrons que chaque nœud n vérifie :

$\mathrm{MCA}'(\mathsf{children}(n), \mathsf{invars}(\mathsf{label}(n)))$.

— Si $n \in \eta - \{n_0\}$ alors, la propriété découle de (2).

— Si $n = (d\ t_{01}\ \ldots\ t_{0p})$ la propriété découle de (6).

Pour montrer (C_2), nous montrons que chaque nœud n vérifie $\mathrm{NO}'(\mathsf{children}(n))$ et que les entrées des conclusions des différents nœuds ne se recouvrent pas deux à deux. Commençons par NO′ :

— Si $n \in \eta - \{n_0\}$ la propriété découle de (3).

— Si $n = (d\ t_{01}\ \ldots\ t_{0p})$ la propriété découle de (7).

Le non recouvrement est vrai pour les nœuds qui appartiennent à l'ensemble $\eta - \{n_0\}$ car la propriété NO est vérifiée pour $\mathsf{Reltree}(\eta)$. Il reste à vérifier que pour tout nœud $n \in \eta - n_0$, les entrées de n et de $(d\ t_{01}\ \ldots\ t_{0p}, r_k)$ ne se recouvrent pas. D'après (H), on sait que les entrées de $(d\ t_{01}\ \ldots\ t_{0p}, r_k)$ ne se recouvrent qu'avec celles de n_0, qui est exclu de l'ensemble $\eta - \{n_0\}$.

2. $\forall n_i \in \eta, \exists t_{i1}\ \ldots\ t_{ip}, \exists \mathsf{Nodes}_i,$
$\begin{cases} n_i = (d\ t_{i1}\ \ldots\ t_{ip}, \mathsf{Nodes}_i) \\ \mathsf{in}(d\ t_{i1}\ \ldots\ t_{ip}, m) \text{ et } \mathsf{in}(d\ t_1\ \ldots\ t_p, m) \text{ non unifiables} \end{cases}$ (H')
Soit $R = \mathcal{C}_{prems}(\emptyset,\ \tau,\ d\ t_1\ \ldots\ t_p,\ \mathsf{invars}(d\ t_1\ \ldots\ t_p, m))$.

La précondition de $\mathcal{C}_{prems}$ est trivialement vérifiée (le premier argument est l'ensemble vide).

D'après la postcondition de $\mathcal{C}_{prems}$, on a donc :

$\forall r_k \in R, \mathrm{MCA}'(r_k, \mathsf{invars}(d\ t_1\ \ldots\ t_p, m))$ $(2')$

et $\forall r_k \in R, \mathrm{NO}'(r_k)$ $(3')$

Le résultat de $\mathcal{C}_{init}$ dans ce cas est $\{\mathsf{Reltree}(\eta \cup \{(d\ t_1\ \ldots\ t_p, r_k)\}) | r_k \in R\}$.

Montrons que $\forall r_k \in R, \mathrm{MCA}(\mathsf{Reltree}(\eta \cup \{(d\ t_1\ \ldots\ t_p, r_k)\}))$ (C_1')

et $\forall r_k \in R, \mathrm{NO}(\mathsf{Reltree}(\eta \cup \{(d\ t_1\ \ldots\ t_p, r_k)\}))$ (C_2').

Soit $r_k \in R$. Montrons que $\mathrm{MCA}(\mathsf{Reltree}(\eta \cup \{(d\ t_1\ \ldots\ t_p, r_k)\}))$. D'après (1), on a $\forall n \in \eta, \mathrm{MCA}'(n, \mathsf{invars}(\mathsf{label}(n), m))$.

Et avec $(2')$, on a $\text{MCA}'(r_k, \mathsf{invars}(d\ t_1\ \ldots\ t_p, m))$. D'où (C'_1).

De la même manière, d'après (1) et $(3')$, on a $\forall n \in \eta \cup \{(d\ t_1\ \ldots\ t_p, r_k)\}, \text{NO}'(n)$.

De plus, le non recouvrement est assuré pour les nœuds qui appartiennent à η. Pour $(d\ t_1\ \ldots\ t_p, r_k)$, on utilise la deuxième partie de (H').

3. Le résultat dans ce cas est $\emptyset$. La postcondition est donc vérifiée.

5.7.5.3 Preuve de correction partielle de $\mathcal{C}_{prems}(\eta, \tau, t, kv)$

Nous considérons la précondition $\text{MCA}'(\eta, kv) \wedge \text{NO}'(\eta)(1)$.

Montrons que $\forall \eta \in \$, \text{MCA}'(\eta, kv) \wedge \text{NO}'(\eta)$.

Pour cela, on distingue 2 cas qui suivent la structure de l'algorithme $\mathcal{C}_{prems}$:

1. $\tau = \emptyset$

   L'algorithme $\mathcal{C}_{concl}$ n'a pas de précondition. D'après la postcondition de $\mathcal{C}_{concl}$, on peut directement déduire celle de $\mathcal{C}_{prems}$.

2. $\tau \neq \emptyset$

   Dans ce cas, le résultat renvoyé par $\mathcal{C}_{prems}$ est $\bigcup_{T \in \tau}(\mathcal{C}_{prem}(\eta, T, \tau - \{T\}, t,\ kv))$.

   Montrons que la précondition de chacun des appels à $\mathcal{C}_{prem}$ est vérifiée. Montrons donc $\text{MCA}'(\eta, kv)$ et $\text{NO}'(\eta)$, propriétés vraie d'après (1). La postcondition de $\mathcal{C}_{prem}$ est donc vérifiée, et permet de déduire les deux propriétés à prouver.

5.7.5.4 Preuve de correction partielle de $\mathcal{C}_{prem}(\eta, T, \tau, t,\ kv)$

Nous considérons la précondition $\text{MCA}'(\eta, kv) \wedge \text{NO}'(\eta)(1)$.

Montrons que $\forall \eta \in \$, \text{MCA}'(\eta, kv) \wedge \text{NO}'(\eta)$.

Pour cela, on distingue 3 cas qui suivent la structure de l'algorithme $\mathcal{C}_{prem}$:

1. $\tau \neq \emptyset \wedge \exists! n_0 \in \eta, \exists \sigma, \mathsf{label}(n_0) = \sigma(T)\ (H)$

   Soit $R = \mathcal{C}_{prems}(\mathsf{children}(n_0),\ \{\sigma(T')|T' \in \tau\},\ \sigma(t),\ kv \cup \mathsf{outvars}(\sigma(T), \mathcal{M}(T)))$.

   Vérifions la précondition de $\mathcal{C}_{prems}$.

   Montrons donc que $\text{MCA}'(\mathsf{children}(n_0), kv \cup \mathsf{outvars}(\sigma(T), \mathcal{M}(T)))$.

   Cela découle de (1), la définition de MCA' et la définition de n_0 $(n_0 \in \eta)$.

   Montrons également que l'on a $\text{NO}'(\mathsf{children}(n_0))$.

Cela vient de (1) (et $n_0 \in \eta$).

Donc la postcondition de $\mathcal{C}_{prem}$ assure que :

$\forall r_k \in R, \mathrm{MCA}'(r_k, kv \cup \mathsf{outvars}(\sigma(T), \mathcal{M}(T)))$ (2)

et $\forall r_k \in R, \mathrm{NO}'(r_k)$ (3).

Dans ce cas le résultat est $\{\{(t_0, r_k)\} \cup (\eta - \{n_0\}) | r_k \in R\}$ avec $t_0 = \mathsf{label}(n_0)$. Il faut donc montrer que $\forall r_k \in R, \mathrm{MCA}'(\{(t_0, r_k)\} \cup (\eta - \{n_0\}), kv)$ (C_1)

et $\forall r_k \in R, \mathrm{NO}'(\{(t_0, r_k)\} \cup (\eta - \{n_0\}))$ (C_2).

Soit $r_k \in R$.

D'après (1), on a $\mathrm{MCA}'((\eta - \{n_0\}), kv)$. Il reste à montrer que $\mathrm{MCA}'(\{(t_0, r_k)\}, kv)$. D'après la définition de MCA′, il faut donc vérifier que $\mathsf{invars}(t_0, \mathcal{M}(t_0)) \subseteq kv$ et $\mathrm{MCA}'(r_k, kv \cup \mathsf{outvars}(t_0, \mathcal{M}(t_0)))$. La première propriété découle de $\mathrm{MCA}'(\eta)$ (1) et $n_0 \in \eta$, et la seconde découle de (2) et $\mathsf{label}(n_0) = \sigma(T)$.

Montrons (C_2). Soit $r_k \in R$. De la même manière que précédemment, ceci découle de (1), (3) et (H).

2. $\mathsf{invars}(T, \mathcal{M}(T)) \subseteq kv \wedge \forall n \in \eta,$
$$\begin{cases} \mathsf{rel}(\mathsf{label}(n)) = \mathsf{rel}(T) \\ \mathsf{in}(\mathsf{label}(n), \mathcal{M}(n)) = \mathsf{in}(T, \mathcal{M}(T)) \\ \mathsf{out}(\mathsf{label}(n), \mathcal{M}(n)) \text{ et } \mathsf{out}(T, \mathcal{M}(T)) \text{ non unifiables} \end{cases} \quad (H')$$
Soit $R = \mathcal{C}_{prems}(\emptyset,\ \tau,\ t,\ kv \cup \mathsf{outvars}(T))$.

La précondition de $\mathcal{C}_{prems}$ est trivialement vérifiée car le premier argument de $\mathcal{C}_{prems}$ est l'ensemble vide. On a donc, d'après la postcondition de $\mathcal{C}_{prems}$:

$\forall r_k \in R, \mathrm{MCA}'(r_k, kv \cup \mathsf{outvars}(T))$ $(2')$

$\forall r_k \in R, \mathrm{NO}'(r_k)$ $(3')$

Dans ce cas le résultat est $\{\{(T, r_k)\} \cup \eta | r_k \in R\}$. On doit donc montrer que $\forall r_k \in R, \mathrm{MCA}'(\{(T, r_k)\} \cup \eta, kv)$ (C_1')

et $\forall r_k \in R, \mathrm{NO}'(\{(T, r_k)\} \cup \eta)$ (C_2').

Soit $r_k \in R$.

Montrons (C_1'). D'après (1), on a $\mathrm{MCA}'(\eta, kv)$.

Il reste donc à vérifier que $\mathrm{MCA}'(\{(T, r_k)\}, kv)$. Cela découle de la définition de MCA′, de $(2')$ et de $\mathsf{invars}(T, \mathcal{M}(T)) \subseteq kv$ (H').

Montrons (C_2'). Soit $r_k \in R$. De la même manière que précédemment, ceci découle de (1), $(3')$ et (H').

3. Le résultat dans ce cas est $\emptyset$. La postcondition est donc vérifiée.

5.7.5.5 Preuve de correction partielle de $\mathcal{C}_{concl}(t,\ kv)$

Le terme t est la conclusion d'un constructeur, de la forme $d\ t_1\ \ldots\ d\ t_p$. Montrons que $\forall \eta \in \$, \mathrm{MCA}'(\eta, m) \wedge \mathrm{NO}'(\eta)$.
Pour cela, on distingue 2 cas qui suivent la structure de l'algorithme $\mathcal{C}_{concl}$:

1. $\mathsf{outvars}(t, m) \subseteq kv\ (H)$
   La propriété $\mathrm{MCA}'((d\ t_1\ \ldots\ d\ t_p), kv)$ découle de (H), et $\mathrm{NO}'((d\ t_1\ \ldots\ d\ t_p))$ est vrai car la propriété NO' est toujours vraie pour un nœud feuille.
2. $\neg(\mathsf{outvars}(t, m) \subseteq kv)$
   Ce cas est trivial.

5.8 Génération de code fonctionnel à partir des Reltree

Comme nous l'avons vu dans la section 5.5, nous générons le code dans un langage fonctionnel commun avant de traduire ce code dans le langage cible. Ce langage commun est appelé MFL (pour "Minimal Functional Langage"). Comme il est traduit vers des langages qui possèdent des caractéristiques différentes, il doit être le plus élémentaire possible. Nous donnons la définition de ce langage dans la section 5.8.1, puis nous décrivons la génération de code à partir des Reltree dans la section 5.8.2. Enfin, nous étudions les propriétés que nous pouvons établir pour le code généré dans la section 5.8.3.

5.8.1 Définition du langage intermédiaire MFL

La définition 3 présente la syntaxe de MFL. MFL est un langage fonctionnel qui permet de définir des fonctions, éventuellement récursives, en utilisant le filtrage. Il ne comporte pas d'ordre supérieur ni de **`if`**, car ces derniers ne sont pas nécessaires pour traiter les définitions inductives considérées. Même si le langage CIC de Coq fait partie des langages cibles de notre extraction et que ce dernier ne supporte pas les motifs de filtrage avec des constructeurs imbriqués, nous avons fait le choix de les inclure dans MFL. Nous compilons les filtrages uniquement lors de la transformation de MFL vers le CIC afin d'éviter de

dupliquer du code dans les fonctions OCaml générées.

Définition 3 (Syntaxe du langage MFL)

$$
\begin{array}{lcl}
fun & ::= & \mathsf{fun}\ f(x_1,\ \ldots,\ x_n)\ \rightarrow\ e \\
e & ::= & x \mid f \mid c \mid e_1\ e_2 \mid (e_1,\ \ldots,\ e_n) \\
 & \mid & (\mathsf{match}\ e\ \mathsf{with} \mid pat_1 \rightarrow e_1\ \ldots \mid pat_n \rightarrow e_n) \\
pat & ::= & x \mid c \mid c\ (pat_1,\ \ldots,\ pat_n) \mid (pat_1,\ \ldots,\ pat_n)
\end{array}
$$

où f est le nom d'une fonction et c le constructeur d'un type inductif. Les deux sont toujours complètement appliqués.

La sémantique de MFL est présentée dans l'annexe A. Les filtrages ne sont pas nécessairement exhaustifs. Si aucun motif ne filtre un terme, le résultat du filtrage sera fail. Même si la propriété 3 assure que le code produit à partir d'un Reltree ne comporte que des motifs de filtrage qui ne se recouvrent pas, les motifs de filtrage sont ordonnés dans MFL, cela en prévision d'extensions, telles que celle décrite dans la section 5.9.4 plus loin dans ce chapitre.

5.8.2 Algorithme de génération de code

L'algorithme de génération de code pour un Reltree r (notée $[\![r]\!]$) de la forme :
$\mathsf{Reltree}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1),\ \ldots,\ (d\ t_{n1}\ \ldots\ t_{np}, \mathsf{Nodes}_n)\})$
et construit à partir d'une relation inductive d dont on extrait la fonction f_d avec
le mode $m = \{i_1,\ \ldots,\ i_m\}$, débute ainsi :

$$
\begin{array}{l}
[\![\mathsf{Reltree}(\{(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}_1),\ \ldots,\ (d\ t_{n1}\ \ldots\ t_{np}, \mathsf{Nodes}_n)\})]\!] \triangleq \\
\quad \mathsf{fun}\ f_d\ x_1\ \ldots\ x_m\ \rightarrow \\
\qquad \mathsf{match}\ (x_1,\ \ldots,\ x_m)\ \mathsf{with} \\
\qquad \mid \mathsf{in}(d\ t_{11}\ \ldots\ t_{1p}, m)\ \rightarrow\ [\![\mathsf{Nodes}_1]\!] \\
\qquad \mid\ \ldots \\
\qquad \mid \mathsf{in}(d\ t_{n1}\ \ldots\ t_{np}, m)\ \rightarrow\ [\![\mathsf{Nodes}_n]\!]
\end{array}
$$

Cela permet de générer le premier filtrage de la fonction extraite. Pour chaque nœud, on génère ensuite le code selon le schéma suivant :

$$
[\![\mathsf{Nodes}]\!] \triangleq
\begin{cases}
\text{si } \mathsf{Nodes} \text{ est } d\ t_1\ \ldots\ t_p \text{ alors } \mathsf{out}(d\ t_1\ \ldots\ t_p, m) \\
\text{si } \mathsf{Nodes} \text{ est } \{(T_{j1}, \mathsf{Nodes}_{j1}),\ \ldots,\ (T_{jn}, \mathsf{Nodes}_{jn})\} \text{ alors} \\
\quad \mathsf{match}\ [\![T_{j1}]\!]\ \mathsf{with} \\
\qquad \mid \mathsf{out}(T_{j1}, \mathcal{M}(T_{j1}))\ \rightarrow\ [\![\mathsf{Nodes}_{j1}]\!] \\
\qquad \mid \ldots \\
\qquad \mid \mathsf{out}(T_{jn}, \mathcal{M}(T_{jn}))\ \rightarrow\ [\![\mathsf{Nodes}_{jn}]\!]
\end{cases}
$$

où $[\![T_{j1}]\!]$ est un appel à la fonction générée à partir de la relation inductive qui apparaît dans T_{j1}. Si $T_{j1} = d_i\ t_1\ \ldots\ t_{p_i}$ alors on a $[\![T_{j1}]\!] = f_{d_i}\ \mathsf{in}(T_{j1}, \mathcal{M}(T_{j1}))$ où f_{d_i} est le nom de la fonction extraite à partir de la relation inductive d_i avec le mode $\mathcal{M}(T_{j1})$. On remarque que T_{j1} a été utilisé dans la construction **`match`**, alors qu'il ne joue aucun rôle particulier par rapport aux $T_{jk},\ \forall k \in 2..n$. La propriété 3 assure que tous ces termes sont égaux. On peut donc choisir n'importe lequel.

5.8.3 Propriétés du code obtenu

Le corps d'une fonction MFL est constitué d'une imbrication de filtrages. C'est donc sur ces derniers que l'on veut établir certaines propriétés, afin que la fonction extraite puisse être traduite vers d'autres langages fonctionnels. Trois propriétés sont en général recherchées sur les filtrages : l'exhaustivité du filtrage, l'absence de cas inutiles, et la linéarité des motifs de filtrage.

D'après la propriété 3 sur les Reltree, on sait que les sorties de prémisses des nœuds qui partagent le même père ne se recouvrent pas (deux à deux). De plus, ce sont ces sorties qui forment les motifs d'un même filtrage. On peut en déduire que les motifs d'un même filtrage ne se recouvrent pas deux à deux. Il ne peut donc pas y avoir de cas inutile et l'ordre des motifs dans le filtrage n'a donc pas d'importance.

En ce qui concerne la linéarité, MFL autorise les motifs de filtrages non linéaires. Il appartient donc au traducteur de MFL vers le langage cible de tenir compte de ce problème si jamais le langage cible ne supporte pas les filtrages non linéaires. De plus, lorsqu'une variable déjà liée par un précédant filtrage apparait dans un motif de filtrage, il ne s'agit pas d'une redéfinition de la variable mais d'un test d'égalité.

Enfin, l'exhaustivité des filtrages n'est pas assurée. En effet, la sémantique de MFL

spécifie que le filtrage renvoie fail lorsqu'aucun motif ne convient. Pour traduire un filtrage MFL vers un langage qui possède un mécanisme d'exceptions comme OCaml, on peut par exemple ajouter un cas par défaut tel que _ → `assert false` à tous les filtrages. En revanche, pour traduire MFL vers un langage qui ne possède pas les exceptions comme le CIC, il est nécessaire de modifier plus profondément la fonction, par exemple en changeant le type de retour par un type option lorsque le filtrage n'est pas exhaustif.

Le langage MFL peut être typé avec un système de types classique. La conjecture décrite dans la propriété 5 exprime qu'à partir d'une définition inductive bien typée, on peut extraire une fonction MFL elle-même bien typée à condition que l'on puisse construire un Reltree valide pour la relation inductive et le mode d'extraction considéré. Pour prouver cette conjecture, il faut utiliser la propriété 2 pour montrer que seules des variables liées sont utilisées dans les calculs, et la propriété 3 pour montrer que les filtrages sont bien formés. Le typage de d permet de vérifier que les termes qui sont traduits directement vers MFL (arguments des prémisses et conclusions) sont bien typés.

Propriété 5 (Conjecture)
Soit une relation inductive d et un mode d'extraction m. S'il est possible de trouver un Reltree r pour d avec le mode m vérifiant $\mathsf{VAL}(r)$ *et si d est bien typée, alors la fonction MFL générée à partir de d ($[\![r]\!]$) est également bien typée.*

La propriété 1 peut être utilisée pour montrer la préservation sémantique entre la relation inductive et la fonction MFL. Il faudrait pour cela donner une sémantique d'exécution des Reltree.

5.9 Extensions

Nous décrivons dans cette section les extensions apportées à l'algorithme d'extraction à partir de relations inductives décrit dans les sections précédentes. Nous pouvons classer ces extensions en deux catégories :

1. celles qui concernent l'ajout d'éléments syntaxiques aux relations inductives,
2. celles qui permettent de lever des restrictions qui proviennent du processus d'extrac-

tion.

En ce qui concerne la première catégorie, nous ajoutons la possibilité d'utiliser des fonctions dans la définition des relations inductives ainsi qu'un mécanisme pour traiter les connecteurs logiques (et, ou et non) dans les prémisses des constructeurs. La seconde catégorie concerne des extensions développées pour résoudre des problèmes particuliers liés au prédicat d'égalité et à la détection du déterminisme des fonctions extraites.

5.9.1 Fonctions utilisées dans la définition des relations inductives

Dans les sections précédentes, nous avons éliminé les appels de fonctions des relations inductives (voir section 5.1). Néanmoins, ce critère est trop restrictif car il impose que tout le développement soit réalisé en utilisant uniquement le style relationnel. Or, pour manipuler un environnement dans le cadre de la sémantique d'un langage, par exemple, on préfère souvent écrire des fonctions. Il existe certains cas dans lesquels la présence de fonctions ne pose aucun problème : dans une sortie d'une conclusion ou bien dans une entrée d'une prémisse, car il suffit de traduire ces termes par des appels de fonction dans le langage MFL. La fonction `plus` est par exemple utilisée dans la sortie de la conclusion du constructeur `cgen` (si l'on considère l'extraction d'une fonction à partir de `fibo` avec le mode {1}).

```
Inductive fibo : nat → nat → Prop :=
| cgen  : forall (a n r1 r2:nat),
          fibo n r1 → fibo (S n) r2 →
          a = S (S n) → fibo a (plus r1 r2)
| cbase : forall (a:nat),
          (a = O) \/ (a = S O) → fibo a (S O).
```

FIGURE 5.11 – Spécification de la relation inductive fibo

En revanche, lorsqu'un appel de fonction est présent dans les sorties d'une prémisse ou bien les entrées d'une conclusion, cela pose davantage de problèmes car nous utilisons ces termes pour établir le déterminisme de la fonction extraite. Pour ce faire, nous les traduisons vers des motifs de filtrage, qui ne doivent pas se recouvrir. Vérifier le non recouvrement est un problème difficile quand il s'agit de valeurs renvoyées par des fonctions.

```
Inductive fibo' : nat → nat → Prop :=
| cgen'  : forall (a n r1 r2:nat),
           fibo' n r1 → fibo' (S n) r2 →
           a = S (S n) → fibo' a (plus r1 r2)
| cbase1 : forall (a:nat),
           a = O → fibo' a (S O)
| cbase2 : forall (a:nat),
           a = S O → fibo' a (S O).
```

FIGURE 5.12 – Spécification équivalente de la relation inductive fibo

5.9.2 Connecteurs logiques

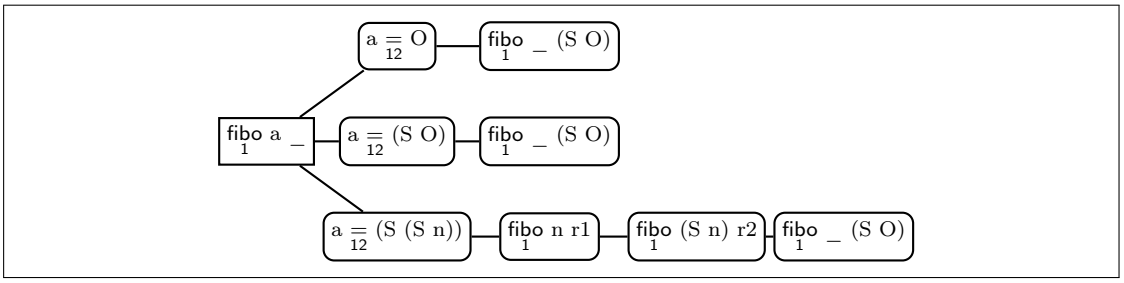
Il est possible d'utiliser des connecteurs logiques, $\wedge$ (et), $\vee$ (ou) et $\neg$ (non), pour écrire des spécifications Coq. C'est par exemple le cas d'une sémantique définie dans MLComp-Cert [Dargaye 2009], et présentée dans l'annexe B. Ces connecteurs logiques ne font pas partie de la définition des relations inductives donnée dans la section 5.1. Néanmoins, il existe un mécanisme pour produire un Reltree à partir d'une spécification qui comporte des connecteurs logiques, comme celle de la figure 5.11.

Une prémisse qui contient un $\wedge$ peut être remplacée par deux prémisses reliées par $\rightarrow$. Pour une prémisse qui contient un $\vee$, on peut modifier la relation inductive en dupliquant le constructeur qui contient le $\vee$. Ces deux constructeurs seront identiques en dehors du fait que l'un des deux contiendra la partie gauche du $\vee$ et l'autre la partie droite. On peut par exemple transformer la relation inductive `fibo` comme cela est fait dans la figure 5.12.

Dans le cas de connecteurs $\wedge$ et $\vee$ imbriqués, on commence par mettre chaque prémisse qui contient des connecteurs logiques sous forme normale conjonctive. Une fois la forme normale conjonctive trouvée, chaque connecteur $\wedge$ peut être remplacé par le symbole $\rightarrow$. Pour les connecteurs $\vee$, nous pouvons, comme indiqué au paragraphe précédent, dupliquer les constructeurs, mais il est préférable de faire cette duplication au dernier moment : lorsque l'on insère la prémisse qui contient le $\vee$ dans le Reltree. Ainsi on peut se contenter de dupliquer uniquement les prémisses qui n'ont pas encore été insérées dans le Reltree plutôt que de dupliquer le constructeur en entier.

La figure 5.13 montre le Reltree construit lors de l'extraction de la fonction `fibo1` à

partir de la relation inductive `fibo` avec le mode {1}, ainsi que la fonction extraite.




```
let rec fibo1 p1 =
  match p1 with
  | a →
    (match a with
     | S S n → let r2 = fibo1 (S n) in
       let r1 = fibo1 n in
       plus r1 r2
     | O → S O
     | S O → S O)
```

FIGURE 5.13 – Reltree et fonction générés à partir de la relation inductive fibo

5.9.3 Utilisation de l'égalité

La relation d'égalité peut jouer plusieurs rôles dans la définition d'une relation inductive suivant le mode avec lequel elle est utilisée. En mode $\{1, 2\}$, lorsque tous ses arguments sont instanciés, elle permet de tester une propriété sur les variables que contiennent ses arguments. C'est par exemple le cas dans le constructeur `cbase` de la figure 5.11. Avec le mode {1} ou le mode {2}, elle permet de définir une variable comme dans le constructeur `cgen` de `fibo`, c'est parfois l'équivalent d'un **`let ... in`** dans le style fonctionnel.

Le prédicat d'égalité peut être utilisé avec les deux rôles dans une même relation inductive. Il est donc peu raisonnable de demander à l'utilisateur de fournir son mode d'extraction comme il en fournit pour les autres relations. Il est donc nécessaire de proposer un mécanisme d'inférence de mode pour ce prédicat en particulier.

Lorsque l'égalité est utilisée dans une spécification, nous avons vu qu'elle peut être employée avec les modes {1}, {2} ou $\{1, 2\}$. Étant donné l'algorithme de construction de Reltree, il n'y a pas besoin d'implanter une vraie inférence de mode pour l'égalité. En effet, chaque fois que l'on rencontre une égalité, on va tenter de construire trois arbres au lieu

d'un, un pour chacun des modes. Et si un mode n'est pas utilisable, l'algorithme finira rapidement par renvoyer un ensemble vide de Reltree pour ce mode. Si plusieurs modes sont possibles, l'ensemble des résultats valides sera conservé.

Une optimisation de la génération de code MFL permet de simplifier la fonction produite lorsque l'égalité est utilisée avec le mode {1} ou {2}. Dans le cas général, pour chaque prémisse de la forme $d_i\ t_1\ \ldots\ t_p$, on génère un terme $f_{d_i}\ t_{m_{i1}}\ \ldots\ t_{m_{in}}$ où f_{d_i} est une fonction extraite à partir de la relation inductive d_i avec le mode m_i. Dans le cas de l'égalité (uniquement avec les modes {1} et {2}), la fonction f_{d_i} est l'identité. On peut alors se passer de cette fonction dans la fonction extraite.

5.9.4 Non déterminisme

Nous avons mentionné à plusieurs reprises que nous ne nous intéressions qu'aux spécifications déterministes. Pour cela, nous avons imposé la propriété 3 sur les Reltree qui permet de s'assurer que les différents motifs d'un filtrage dans la fonction générée ne se recouvrent pas. Néanmoins, cette restriction est parfois handicapante.

Par exemple, on peut étendre la spécification présentée dans la section 5.4.1, afin d'obtenir la spécification de la figure 5.14. Cette spécification permet de renvoyer le résultat `NotFound` quand l'entier recherché ne se trouve pas dans l'arbre alors que la précédente renvoyait un chemin quelconque se terminant par `NotFound`. Dans cette spécification, les constructeurs `Search_sup` et `Searchsup_nf` se recouvrent (ainsi que `Search_inf` et `Searchinf_nf`). Il serait possible d'écrite cette spécification de manière déterministe, mais cela obligerait à ajouter quatre constructeurs supplémentaires pour énumérer les constructeurs du type `path`.

Il existe une solution pour traiter la spécification de la figure 5.14 consistant à autoriser certains motifs de filtrage qui se recouvrent. Si un motif de filtrage a est strictement plus général d'un autre motif b (a et b sont ici des noms arbitraires), alors on peut autoriser le fait que a et b apparaissent dans le même filtrage, et on placera b avant a. Dans l'exemple de la figure 5.14, les constructeurs `Searchsup_nf` et `Searchinf_nf` contiennent des motifs respectivement plus généraux que ceux des constructeurs `Search_sup` et `Search_inf`.

```
Inductive search : bst → nat → path → Prop :=
  | Search_empty : forall n, search Empty n NotFound
  | Search_found : forall n m t1 t2, compare n m Eq →
      search (Node t1 m t2) n EndPath
  | Search_inf : forall n m t1 t2 b, search t1 n b →
      compare n m Inf → search (Node t1 m t2) n (Left b)
  | Searchinf_nf : forall n m t1 t2, search t1 n NotFound →
      compare n m Inf → search (Node t1 m t2) n NotFound
  | Search_sup : forall n m t1 t2 b, search t2 n b →
      compare n m Sup → search (Node t1 m t2) n (Right b)
  | Searchsup_nf : forall n m t1 t2, search t2 n NotFound →
      compare n m Sup → search (Node t1 m t2) n NotFound.
```

FIGURE 5.14 – Une autre version de la recherche dans un arbre binaire

Nous nous permettons de choisir l'ordre des motifs de filtrage dans ce cas précis car le fait de placer le motif le plus général en premier n'aurait pas de sens : le motif le moins général ne serait jamais utilisé.

Néanmoins, cette extension qui s'écarte du cadre fixé initialement n'est pas activée par défaut. L'activation se fait via une option de la commande d'extraction. Lorsqu'elle est activée, la complétude de la fonction extraite n'est plus garantie.

Plus formellement, décrivons les changements que nécessite cette extension dans les définitions et propriétés données au début de ce chapitre. Nous définissons tout d'abord la notion de motif de filtrage plus général :

Définition 4 (Motif de filtrage plus général)

On définit que t_1 est plus général que t_2 (ce que l'on note $t_1 > t_2$) comme suit :

$$t_1 > t_2 \iff \begin{cases} (t_1 = v \wedge t_2 = c_l\ a_1\ \ldots\ a_{p_l}) \\ \vee (t_1 = c_l\ a'_1\ \ldots\ a'_{p_l} \wedge t_2 = c_l\ a_1\ \ldots\ a_{p_l} \wedge \\ \quad \exists i \in 1..p_l, a'_i > a_i \wedge\ \forall i \in 1..p_l, a'_i > a_i \vee a'_i = a_i) \\ \vee (t_1 = (a'_1,\ \ldots,\ a'_{p_l}) \wedge t_2 = (a_1,\ \ldots,\ a_{p_l}) \wedge \\ \quad \exists i \in 1..p_l, a'_i > a_i \wedge\ \forall i \in 1..p_l, a'_i > a_i \vee a'_i = a_i) \end{cases}$$

où a_i et a'_i sont des termes, au sens de la définition des relations inductives.

Pour pouvoir tenir compte de l'ordre des motifs de filtrage, il faut modifier la définition des Reltree, en utilisant des listes de nœuds à la place des ensembles de nœuds :

Définition 5 (Reltree avec des listes)

On définit Reltree' :

$\mathsf{Reltree'}([(d\ t_{11}\ \ldots\ t_{1p}, \mathsf{Nodes}'_1),\ \ldots,\ (d\ t_{k1}\ \ldots\ t_{kp}, \mathsf{Nodes}'_k)])$

où Nodes'_i *est* $[(T_{i1}, \mathsf{Nodes'}_{i1}),\ \ldots,\ (T_{ik}, \mathsf{Nodes}'_{ik})]$ *ou* $d\ t_{i1}\ \ldots\ t_{ip}$.

Les propriétés 1, 2 et 3 doivent également être modifiées pour correspondre à cette nouvelle représentation. En particulier, la propriété 3 doit évoluer car les motifs ne sont plus nécessairement non unifiables. Pour les propriétés 1 et 2, les changements sont seulement syntaxiques, car les listes dans la définition 5 peuvent être utilisées comme des ensembles.

Pour la propriété 3, nous définissons la relation suivante, afin d'ordonner les nœuds :

Définition 6 (Motif de filtrage plus général ou non unifiable)

On définit que t_1 *est plus général que* t_2 *ou non unifiable avec* t_2 *:*

$t_1 \succ t_2 \iff t_1 > t_2$ *ou* t_1 *et* t_2 *ne sont pas unifiables*

Nous pouvons alors réécrire la propriété 3 :

Propriété 6 (Ordonnancement pour les Reltree')

Soit un Reltree' r pour d. r est ordonné avec le mode m si et seulement si $\mathsf{NO}_l(r)$.

Avec

$$\mathsf{NO}'_l(nodes) \triangleq$$
$$\begin{cases} \textit{si nodes est } d\ t_1\ \ldots\ t_p \textit{ alors } \mathsf{true} \\ \textit{si nodes est } [(T_1, \mathsf{Nodes}_1);\ \ldots;\ (T_k, \mathsf{Nodes}_k)] \textit{ alors} \\ \quad \forall i, i \in i..k, \begin{cases} \mathsf{NO}'_l(\mathsf{Nodes}_i) \wedge \\ \quad \forall j, j \in i..k, i > j \Rightarrow \mathsf{in}(T_i, \mathcal{M}(T_i)) = \mathsf{in}(T_j, \mathcal{M}(T_j)) \wedge \\ \quad \mathsf{out}(T_i, \mathcal{M}(T_i)) \succ \mathsf{out}(T_j, \mathcal{M}(T_j)) \end{cases} \end{cases}$$

L'algorithme de construction des Reltree' est légèrement différent de celui de construction des Reltree. Nous ne détaillons pas ces changements ici, mais il s'agit essentiellement d'ordonner les nœuds qui partagent le même père.

La génération de code à partir d'un Reltree' est, elle, quasiment identique à la génération de code à partir d'un Reltree. La seule différence est qu'au lieu de prendre un ordre quelconque pour les nœuds lorsque l'on construit un filtrage, on prend l'ordre donné par

```
let rec search12 p1 p2 =
  match (p1, p2) with
  | (Empty, n) → NotFound
  | (Node (t1, m, t2), n) →
    (match compare12 n m with
     | Eq → EndPath
     | Inf →
       (match search12 t1 n with
        | NotFound → NotFound
        | b → Left b)
     | Sup →
       (match search12 t2 n with
        | NotFound → NotFound
        | b → Right b))
```

FIGURE 5.15 – Code extrait à partir de la relation inductive de la figure 5.14 avec le mode $\{1, 2\}$

la liste des nœuds. Pour la relation inductive de la figure 5.14, on obtient le code de la figure 5.15.

5.10 Relations mutuellement inductives

Les algorithmes de construction de Reltree et de génération de code ont été présentés dans le cas de l'extraction d'une fonction à partir d'une relation inductive et de ses dépendances. Il est donc possible d'extraire des fonctions qui dépendent d'autres fonctions en commençant par extraire ces dernières. Par exemple, la relation `exec` utilise, dans sa définition, la relation inductive `eval`. Il suffit pour extraire `exec` en mode $\{1, 2\}$ d'avoir déjà extrait `eval` en mode $\{1, 2\}$. Nous demandons pour cette raison à l'utilisateur de fournir une liste de modes lorsqu'il souhaite extraire une fonction à partir d'une relation inductive qui possède des dépendances.

De la même manière, deux relations inductives peuvent dépendre l'une de l'autre. Prenons par exemple les relations inductives de la figure 5.10, qui caractérisent les nombres pairs et impairs. De ces deux relations, on peut extraire les deux fonctions de la figure 5.17. Pour cela, on peut toujours utiliser les algorithmes décrits plus tôt. En effet, il suffit de connaître les modes d'extraction des deux relations inductives, puis de supposer lors de l'extraction de la première que la deuxième est déjà extraite, même si ce n'est pas le cas.

On peut ainsi obtenir deux fonctions MFL, chacune faisant appel à l'autre. La sémantique du langage MFL permet leur exécution : il suffit que les deux fonctions récursives soient présentes dans Δ_f avant l'exécution. À partir de ces deux fonctions MFL seront produites deux fonctions mutuellement récursives en Coq ou en OCaml.

```
Inductive even : nat → Prop :=
  | evenO : even 0
  | evenS : forall n, odd n → even (S n)
with odd : nat → Prop :=
  | oddS : forall n, even n → odd (S n).
```

FIGURE 5.16 – Spécification des relations pair et impair

```
let rec odd_full p1 = match p1 with
  | S n →
    (match even_full n with
     | true → true
     | _ → false)
  | _ → false

and even_full p1 = match p1 with
  | O → true
  | S n →
    (match odd_full n with
     | true → true
     | _ → false)
```

FIGURE 5.17 – Fonctions extraites avec le mode {1} : pair et impair

Chapitre 6

Extraction vers ML

L'implantation de l'extraction de code vers le langage OCaml est réalisée au sein de l'outil d'aide à la preuve Coq. Le schéma de compilation a déjà été donné dans le chapitre précédent (figure 5.7). Un plugin intégré à Coq permet d'extraire des programmes fonctionnels à partir des fonctions et des preuves écrites en Coq. Nous ferons référence à cette extraction en l'appelant "extraction native". Ce plugin peut produire des programmes OCaml ou Haskell, qui par défaut sont affichés sur la sortie standard ou écrits dans un fichier qu'il est ensuite possible de compiler pour obtenir un exécutable. Ce procédé est généralement utilisé pour obtenir une version efficace d'un programme qui s'exécuterait trop lentement dans Coq. Le code produit pourra également être utilisé dans des développements OCaml ou Haskell, sans liaison avec Coq. Notre objectif est ici d'étendre ce mécanisme d'extraction, en ajoutant les relations inductives aux types d'objets à partir desquels le mécanisme d'extraction est capable de produire du code.

Nous commençons par une présentation du plugin d'extraction native dans la section 6.1 puis nous donnons quelques résultats dans la section 6.2 et enfin nous décrivons notre implantation en détail dans la section 6.2.

6.1 Plugin d'extraction native de Coq

L'extraction de code fonctionnel dans Coq existe depuis 1989 [Paulin-Mohring 1989a; Paulin-Mohring and Werner 1993]. Elle a été complètement réécrite depuis la version 7.0 de Coq [Letouzey 2002]. Elle permet notamment d'extraire du code fonctionnel à partir

des types de données inductifs, des fonctions et des constantes qui sont présents dans l'environnement global de Coq [Letouzey 2008]. Cette extraction est récursive et permet de collecter les dépendances. En revanche, les parties purement logiques (telles que les relations inductives), ne sont pas traités par l'extraction et ne figurent donc pas dans le code extrait.

6.1.1 Fonctionnement de l'extraction

Une commande spécifique, **`Extraction`** permet de demander l'extraction d'un terme, le plus souvent une fonction. Il est également possible de spécifier à cette commande si l'on veut ou non extraire automatiquement les dépendances.

Une fois les éventuelles dépendances collectées, l'extraction se passe en deux temps. Tout d'abord, les sous-termes logiques sont effacées dans les éléments à extraire. Ensuite, les constructions Coq, initialement dans le formalisme du CIC, sont traduites vers un langage intermédiaire : MiniML. Le bloc de définitions MiniML est ensuite traduit à son tour en utilisant l'un des deux "pretty printers", qui permettent respectivement de générer du code OCaml ou Haskell. Il existe aussi un troisième "pretty printer" pour Scheme mais il n'est plus maintenu. Une série d'optimisations est réalisée afin de rendre le code plus élégant et plus efficace. En effet le CIC a été conçu pour être minimaliste car c'est le cœur du système Coq et il doit être facilement vérifiable. Ainsi, il n'y a pas de construction pour **`if`**, par exemple, même s'il est présent dans la syntaxe de Coq. Le **`if`** est en fait représenté par un filtrage en Coq. Ces filtrages sont eux-mêmes transformés en tests par le plugin d'extraction.

6.1.2 Langage intermédiaire

Le langage intermédiaire utilisé par le plugin d'extraction native est nommé MiniML. Il s'agit d'un langage fonctionnel très basique. Lors du développement de la première implantation par David Delahaye, MiniML avait été enrichi, notamment avec les gardes pour les motifs de filtrage, les tuples, et d'autres constructions. Dans l'implantation actuelle, nous avons réduit le nombre de constructions nécessaires. Une nouvelle version de MiniML a été conçue avec Pierre Letouzey, qui est responsable du développement du plugin d'extraction native de Coq. Cette version a été adoptée dans Coq depuis la version 8.4. Elle est conçue

pour permettre la génération de code vers ce langage, tout en minimisant les changements par rapport à la version précédente. Les tuples et les motifs de filtrage composés ont été ajoutés afin d'éviter de multiplier les filtrages.

6.2 Implantation

Nous apportons à Coq un mécanisme permettant l'extraction de comportements calculatoires présents dans des relations inductives sous forme de programmes OCaml. Pour cela, nous utilisons l'environnement décrit dans le chapitre 5, et nous ajoutons une étape pour traduire le code MFL vers OCaml. Le langage MFL étant très facile à traduire vers OCaml, nous aurions pu nous contenter de cette traduction. Néanmoins, le fait de traduire MFL vers MiniML offre plusieurs avantages.

Tout d'abord, cela permet aux deux mécanismes d'extraction d'interagir. Les éléments qui sont extraits avec l'extraction native et avec l'extraction des relations inductives partagent le même fichier de sortie. Il est donc possible d'utiliser des fonctions dans une relation inductive que l'on souhaite utiliser pour l'extraction. Mieux encore, il est possible de demander, lors de l'extraction d'une relation inductive, l'extraction simultanée de toutes ses dépendances : types de données inductifs, fonctions, constantes, ... L'extraction des relations inductive créé alors une fonction MiniML qui contient des références explicites à toutes les dépendances qu'elle utilise. Elle donne ensuite cette fonction à l'extraction native qui se charge de collecter récursivement toutes les dépendances de cette fonction. Enfin, l'extraction native utilise un "pretty printer" pour traduire toutes les fonctions et leurs dépendances.

Cela permet également de profiter des différents "pretty printers" de l'extraction native, actuellement, OCaml et Haskell. Cela permet aussi d'améliorer sensiblement la qualité du code produit en profitant des optimisations qui sont réalisées par l'extraction native. Certains filtrages sont transformés en tests et d'autres en définition locales (`let in`).

Parmi les trois implantations que nous avons réalisées (les chapitres qui suivent traitent de l'extraction vers Coq et vers Focalize), l'extraction vers OCaml est celle qui permet d'extraire le plus grand nombre de relations inductives car pratiquement toutes les extensions

envisagées ont été implantées. Ainsi, il est possible d'extraire des fonctions mutuellement récursives, des relations inductives qui utilisent le prédicat d'égalité et des connecteurs logiques. L'extraction vers OCaml permet notamment d'extraire l'ensemble des exemples présentés dans le chapitre 5.

6.2.1 De MFL à MiniML

Le schéma de compilation vers OCaml (figure 5.7) est dérivé du schéma de compilation général (figure 5.6). Les deux étapes ajoutées, qui concernent spécifiquement Coq sont les étapes C0 et C1. Les parties en pointillé, ainsi que les étapes C2 et C3 appartiennent à l'extraction native de Coq. L'étape C0 consiste à normaliser une relation inductive de Coq tout en vérifiant qu'elle est conforme avec les limitations imposées dans la définition de la section 5.1. La génération de code MiniML à partir du code MFL (étape C1) est une simple traduction puisque MiniML inclut le langage MFL. Néanmoins, MiniML requiert certaines informations sur les termes originaux (dans le formalisme du CIC) et nous conservons donc une table pour retrouver les termes originaux à partir des identificateurs (nom de variables, de fonctions et de constantes) qui sont présents dans les Reltree et les fonctions MFL.

Nous avons présenté des fonctions extraites vers OCaml dans le chapitre 5, comme dans la figure 5.15 par exemple.

6.2.2 Limites

La limite la plus importante se trouve dans l'interaction entre l'extraction native et notre extraction à partir de relations inductives. Si l'extraction à partir de relations inductives peut faire appel à l'extraction native pour les dépendances, le contraire n'est pas vrai. Il n'est donc pas possible d'utiliser une fonction extraite à partir d'une relation inductive dans une seconde fonction extraite vers OCaml. Cette limitation est due à deux causes.

Premièrement, l'extraction des relations inductives nécessite que des modes d'extraction soient fournis par l'utilisateur. Une solution envisageable consisterait à inférer automatiquement le mode d'extraction.

La seconde difficulté est plus difficile à résoudre. En effet, les fonctions extraites à partir de relations inductives vers OCaml n'existent pas dans l'environnement de Coq. Il

est impossible d'écrire une fonction dans Coq faisant référence à ces dernières.

Deux solutions peuvent néanmoins être mises en place pour utiliser une fonction extraite vers OCaml à partir d'une relation inductive dans une autre fonction. La première consiste à écrire la fonction en question en OCaml dans le fichier extrait ou dans un second fichier. La seconde, à utiliser l'extraction vers Coq. Nous décrivons cette possibilité dans le chapitre suivant.

Chapitre 7

Extraction vers Coq

L'extraction vers Coq suit le même schéma que l'extraction vers OCaml. En revanche, le CIC, langage interne de Coq est plus restrictif qu'un langage fonctionnel classique puisqu'il est nécessaire de prouver la terminaison de toutes les fonctions et que tous les filtrages doivent être exhaustifs. De plus, la version interne du CIC vers laquelle nous compilons ne supporte que les filtrages de tête (c'est-à-dire non imbriqués). Il est donc nécessaire d'ajouter une étape de compilation à partir du langage MFL pour obtenir un code fonctionnel de plus bas niveau.

Nous commençons par décrire brièvement la logique sous-jacente de Coq, à savoir le CIC dans la section 7.1. Nous décrivons ensuite la compilation vers le CIC dans la section 7.2, puis nous discutons des problèmes de terminaison dans la section 7.3. Coq ne disposant pas de mécanisme d'exception, les fonctions qui ne sont pas totales sont complétées en utilisant un type `option`. Nous détaillons les problèmes liés aux fonctions non totales dans la section 7.4, Puis nous donnons quelques résultats dans la section 7.5. Enfin, nous abordons la génération automatique des preuves de correction des fonctions extraites dans la section 7.6.

7.1 Calcul des constructions inductives de Coq

Tout objet Coq (fonction, formule, preuve) est représenté en interne par un terme du calcul des constructions inductives (CIC) [Coquand and Huet 1988; Werner 1994]. Ainsi, le même formalisme logique permet de décrire à la fois les parties logiques et les parties

exécutables (fonctions). Commençons par décrire les éléments du CIC que nous utilisons dans ce chapitre.

En ce qui concerne la partie logique du CIC qui nous intéresse ici, les spécifications ont déjà été décrites dans la section 5.1 et les preuves font l'objet d'une section dédiée de ce chapitre (section 7.6).

Le sous-ensemble fonctionnel du CIC que nous utilisons est récapitulé dans le tableau 7.1. Nous utilisons les notations du manuel de Coq [development team 2004]). La terminaison des fonctions écrites dans le CIC doit être prouvée. Nous traitons ce point particulier dans la section 7.3. Les filtrages doivent être nécessairement exhaustifs, si bien que tous les constructeurs (c_1, ..., c_n) du type de t_m sont présents.

$$
\begin{array}{l}
t ::= fix\ f\ (x_1 : \tau_1)\ \ldots\ (x_n : \tau_n)\ :\ \tau\ :=\ t \\
\quad \mid\ f\ t1\ \ldots\ t_{p_f} \mid c\ t1\ \ldots\ t_{p_c} \mid let\ x\ :=\ t_1\ in\ t_2 \mid x \\
\quad \mid\ (match\ t_m\ \ with\ c_1\ x_{11}\ \ldots\ x_{1p_1}\ \Rightarrow\ f_1\ \mid \\
\quad\ \ldots \mid c_n\ x_{n1}\ \ldots\ x_{np_n}\ \Rightarrow\ f_n) \\
\qquad\qquad \text{où } \tau_i \text{ est un type.}
\end{array}
$$

TABLE 7.1 – Sous-ensemble du langage CIC

7.2 Compilation vers Coq

La génération de code Coq à partir de relations inductives, présentée dans la figure 7.1, est similaire à la génération de code OCaml. Cependant, une étape supplémentaire est nécessaire afin d'obtenir des filtrages compatibles avec le CIC. Dans un premier temps, nous traduisons le langage MFL vers une version étendue du fragment fonctionnel du CIC, que nous nommons eCIC (étape C4).

Ce langage est identique au CIC, en dehors du fait qu'il possède des filtrages plus complexes (tuples et constructeurs de types inductifs imbriqués). La traduction de MFL vers eCIC est triviale, tout comme dans le cas de l'étape C1 (figure 5.7) pour la compilation de MFL vers MiniML. La principale difficulté se trouve dans la compilation des filtrages (C5), qui permet de passer du eCIC au CIC. Coq possède un algorithme de compilation des filtrages, mais nous implantons notre propre algorithme afin d'avoir un contrôle plus fin sur cet algorithme, ce qui facilite la génération des preuves.

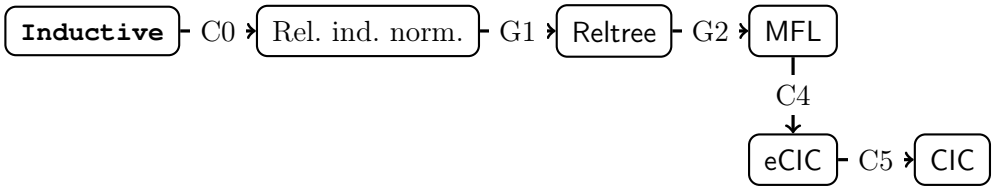


FIGURE 7.1 – Schéma de compilation vers Coq

7.2.1 Compilation des filtrages

Tout d'abord, nous donnons une définition du langage intermédiaire eCIC. Il s'agit du même langage que le CIC, à l'exception près que les filtrages ont la forme suivante :

$match\ (t_{m1},\ \ldots,\ t_{mn})\ with$

$\quad \mid p_{11},\ \ldots,\ p_{1n}\ \Rightarrow\ t_1 \mid\ \ldots\ \mid p_{k1},\ \ldots,\ p_{kn}\ \Rightarrow\ t_k$

où

$p\ ::=\ c\ p_1\ \ldots\ p_{p_c} \mid x \mid \_$

Nous nous plaçons également dans le cas de motifs de filtrage dans lesquels chaque variable n'apparaît qu'une seule fois.

7.2.1.1 Spécifications complètes

Nous nous plaçons ici dans le cas où tous les filtrages présents dans la fonction MFL sont complets. Il n'est alors pas nécessaire de compléter les filtrages de la fonction extraite à l'aide de cas par défaut. On parle alors de fonction complète, et on dit également que la relation inductive d qui, extraite avec le mode m, a engendré cette fonction est complète pour le mode m. Le cas des spécifications non complètes est traité dans la section qui suit. Notons que les spécifications extraites en mode complet (tous les arguments sont des entrées), ne sont généralement pas des spécifications complètes pour ce mode. Sinon, la fonction extraite renverrait toujours `true`.

Notre algorithme de compilation des filtrages, appelé $\mathcal{D}$ dans la suite, est une adaptation d'une technique standard de compilation des filtrages [Le Fessant and Maranget 2001], en utilisant des matrices de motifs de filtrage que nous compilons progressivement, colonne par colonne. L'algorithme $\mathcal{D}$ prend en paramètre un terme et non une matrice, mais il est possible de représenter les motifs de filtrage (p_{ij}) sous forme matricielle. La fonction

select_rows utilise la représentation matricielle. L'algorithme $\mathcal{D}$ distingue quatre cas :

1. s'il ne reste qu'une ligne ($k = 1$) et plus aucun motif ($n = 0$), on produit le terme qui correspond à la ligne restante,
2. sinon, si une variable est présente dans la première colonne, on remplace toutes les variables présentes dans la première colonne par une variable fraîche (dans t_i) ou par un _ (dans la première colonne) et on produit un *let in*.
3. sinon, si la première colonne contient au moins un constructeur, on produit un filtrage sur le premier terme à filtrer (m_1)
4. sinon, la première colonne ne contient que des _ et on supprime cette colonne.

$\mathcal{D}(\mathsf{match}\ (m_1,\ \ldots,\ m_n)\ \mathsf{with}\ p_{11},\ \ldots,\ p_{1n} \to t_1 \mid \ldots \mid p_{k1},\ \ldots,\ p_{kn} \to t_k) \triangleq$

1. si n est 0 et k est 1 alors on produit le code : $\mathbf{t_1}$

2. sinon, si $\exists i \in 1..k, p_{i1}$ est une variable alors
soit v_f une variable fraîche
$\forall i \in 1..k,$ on définit p'_{i1} et t'_i comme suit :
si p_{i1} est une variable v_i
$\begin{cases} p'_{i1} = \_ \\ t'_i = \sigma(t_i) \text{ avec } \sigma = (v_i, v_f) \end{cases}$
sinon
$\begin{cases} p'_{i1} = p_{i1} \\ t'_i = t_i \end{cases}$
On produit le code suivant :
let $\mathbf{v_f} = \mathbf{m_1}$ in
$\mathcal{D}(\mathsf{match}\ (v_f,\ m_2,\ \ldots,\ m_n)\ \mathsf{with}$
$\quad p'_{11},\ \ldots,\ p'_{1n} \to t'_1 \mid \ldots \mid p'_{k1},\ \ldots,\ p'_{kn} \to t'_k)$

3. sinon, si $\exists i \in 1..k, p_{i1}$ soit de la forme $c\ a_1\ \ldots\ a_p$ alors
soit τ le type de m_1
et soit $\{c_1,\ \ldots,\ c_l\}$ les constructeurs du type τ
On produit le code suivant :
match $\mathbf{t_{m1}}$ with
| **$\mathbf{c_1}$ $\mathbf{t_{11}}$ $\ldots$ $\mathbf{t_{1p}}$** $\to \mathcal{D}(\mathsf{select\_rows}$
$\quad (c_1\ a_{11}\ \ldots\ a_{1p},\ \mathsf{match}\ (a_{m1},\ \ldots,\ a_{mn})\ \mathsf{with}$
$\quad\ p_{11},\ \ldots,\ p_{1n}\ \to\ t_1 \mid \ldots \mid p_{k1},\ \ldots,\ p_{kn}\ \to\ t_k))$
$\ldots$
| **$\mathbf{c_l}$ $\mathbf{a_{l1}}$ $\ldots$ $\mathbf{a_{lp}}$** $\to \mathcal{D}(\mathsf{select\_rows}$
$\quad (c_l\ a_{l1}\ \ldots\ a_{lp},\ \mathsf{match}\ (t_{m1},\ \ldots,\ t_{mn})\ \mathsf{with}$
$\quad\ p_{11},\ \ldots,\ p_{1n}\ \to\ t_1 \mid \ldots \mid p_{k1},\ \ldots,\ p_{kn}\ \to\ t_k))$
où les a_{jq} sont des variables fraîches générées pour chaque constructeur

4. sinon $\mathcal{D}(\mathsf{match}\ (m_2,\ \ldots,\ m_n)\ \mathsf{with}\ p_{12},\ \ldots,\ p_{1n} \to t_1 \mid \ldots \mid p_{k2},\ \ldots,\ p_{kn} \to t_k)$

L'algorithme select_rows permet de selectionner les lignes qui sont pertinantes pour un constructeur c donné. Ces lignes sont celles qui contiennent soit le constructeur c soit une variable dans la première colonne.

select_rows($c\ a_1\ \ldots\ a_p$, match $(m_1,\ \ldots,\ m_n)$ with
$p_{11},\ \ldots,\ p_{1n}\ \rightarrow\ t_1 \mid \ldots \mid p_{k1},\ \ldots,\ p_{kn}\ \rightarrow\ t_k) \triangleq$

on produit le code suivant : **match** $(\mathbf{a_1},\ \ldots,\ \mathbf{a_p},\ \mathbf{m_1},\ \ldots,\ \mathbf{m_n})$ **with**
puis on décompose les lignes $()p_{i1},\ \ldots,\ p_{in}\ \rightarrow\ t_i)$ en trois ensembles :
L^c : contient les lignes pour lesquelles p_{i1} est de la forme $c\ a_{i1}\ \ldots\ a_{ip}$ (où c est le constructeur passé en argument à select_rows)
L^* : contient les lignes pour lesquelles p_{i1} est _
L^- : contient les autres lignes
on produit le code suivant pour chaque ligne de L^* :
$\mid$ _, $\cdots$, _, $\mathbf{p_{i2}}$, $\cdots$, $\mathbf{p_{in}}\ \rightarrow\ \mathbf{t_i}$
et on produit le code suivant pour chaque ligne de L^c :
$\mid\ \mathbf{a_{i1}},\ \ldots,\ \mathbf{a_{ip}},\ \mathbf{p_{i2}},\ \ldots,\ \mathbf{p_{in}}\ \rightarrow\ \mathbf{t_i}$

Lorsque le constructeur c est présent dans le premier motif (p_{i1}) d'une ligne, on étend cette ligne avec les aguments de c. Si un autre constructeur (c') est présent, la ligne est ignorée. Enfin, si la ligne a pour premier motif _, elle est étendue avec des _ de manière à avoir le même nombre de colonnes que les autres lignes.

La figure 7.2 montre par exemple, le résultat de la compilation des filtrages de la fonction `search12` (figure 5.4). Les variables portent des noms de la forme `fix_i` car l'algorithme de compilation implanté génère des variables fraîches.

```
Fixpoint search12 (p1 : bst) (p2 : nat) {struct p1} : path :=
  match p1 with
  | Empty ⇒ let fix_24 := p2 in NotFound
  | Node fix_27 fix_26 fix_25 ⇒
      let fix_28 := p2 in
      match compare12 fix_28 fix_26 with
      | Inf ⇒ let fix_29 := search12 fix_27 fix_28 in Left fix_29
      | Sup ⇒ let fix_30 := search12 fix_25 fix_28 in Right fix_30
      | Eq ⇒ EndPath
      end
  end.
```

FIGURE 7.2 – Résultat de la compilation des filtrages de la fonction search12

7.2.1.2 Spécifications non complètes

Lorsque la spécification inductive n'est pas complète, c'est-à-dire que les filtrages ne sont pas exhaustifs, la fonction extraite n'est pas totale. Nous avons choisi dans ce cas de transformer son résultat de type `T` initial en un type optionnel (`option T`) en Coq. Ainsi tous les résultats définis dans la spécification initiale seront introduits à l'aide du constructeur `Some` alors que les cas complétés, absents initialement, se verront associés le résultat `None`. La complétude des fonctions est testée en réalisant une première passe avec l'algorithme $\mathcal{D}$. En effet, dans le cas où l'on compile les filtrages d'un fonction non complète, l'algorithme select_rows ne renvoie parfois aucune ligne. Dans ce cas, l'algorithme $\mathcal{D}$ échoue. Il est alors nécessaire d'ajouter une optimisation à l'algorithme précédant. Selon que le mode d'extraction envisagé (partiel ou complet), le traitement supplémentaire sera différent. Le schéma est néanmoins le même dans les deux cas : on réalise une première fois la compilation avec l'algorithme $\mathcal{D}$, qui échoue lorsque la fonction n'est pas complète. On modifie alors la fonction MFL, puis on rejoue la compilation avec l'algorithme $\mathcal{D}$.

Dans le cas du mode complet (tous les arguments de la relation inductive sont considérés comme des entrées pour la fonction extraite), toutes les sorties de la fonction MFL, tant que cette dernière n'a pas encore été complétée, sont égales à `true`. Dans les autres cas, la fonction extraite doit tout simplement renvoyer `false`. Pour cela, il n'est pas nécessaire de modifier l'algorithme de compilation des filtrages. Il suffit d'ajouter le cas par défaut `| _ → false` à tous les filtrages de la fonction MFL, puis de rejouer l'étape de compilation C4.

Pour le mode partiel, il n'est pas possible d'appliquer un mécanisme identique. En effet, lorsque la relation inductive n'est pas définie, on ne peut pas renvoyer une valeur du bon type pour la fonction extraite. Par exemple, la fonction de soustraction ne renvoie pas toujours de résultat. Comme Coq ne dispose pas de mécanisme d'exceptions, on change le type de retour de la fonction. Ce dernier ne sera plus `nat` mais `option nat` pour la soustraction. Dans tous les cas où la fonction n'est pas définie, elle renvoie `None`. Cela nécessite de modifier la fonction MFL. Tout d'abord, on change le type de retour de la fonction. Dès qu'un filtrage a la forme **`match`** `f_d ...` **`with`** `...` tous les motifs de ce

filtrage doivent être modifiés. On les fait précéder du constructeur `Some`. Ensuite, toutes les sorties de la fonction MFL doivent aussi être précédées du constructeur `Some`. Et enfin, on complète les filtrages incomplets de la fonction avec la clause suivante : `| _ → None`. Une fois la fonction MFL modifiée, on rejoue l'étape de compilation C4.

7.3 Problèmes de terminaison

Une contrainte supplémentaire imposée par Coq est la terminaison des fonctions. Notre objectif concernant l'outil d'extraction est de fournir automatiquement la preuve de terminaison dans le cas de la récurrence structurelle et bien fondée. Le scénario d'extraction est en quelque sorte le même qu'avec la construction **`Function`** [Barthe et al. 2006]. L'utilisateur indique l'argument qui décroît structurellement dans le cas d'une induction structurelle, ou celui qui décroit selon la relation bien fondée qui assure la terminaison, qu'il fournit également. Pour l'instant, seule l'induction structurelle est traitée (section 7.3.1). En contre-partie, nous avons implanté un procédé pour extraire des fonctions sans avoir à fournir leur preuve de terminaison (section 7.3.2).

7.3.1 Récursion structurelle

Dans le cas de la récursion structurelle, nous déléguons à Coq le soin de faire la preuve de terminaison, en lui transmettant le numéro de l'argument décroissant fourni par l'utilisateur dans la commande d'extraction. Les conditions de décroissance sont les mêmes que pour la construction **`Fixpoint`** de Coq. Pour un argument entier par exemple, s'il est déstructuré de la manière suivante : **`match`** `p` **`with`** `S (S n) → ...`, l'appel récursif pourra se faire sur le terme `n` mais pas sur le terme `S n`, à moins d'utiliser la syntaxe suivante : **`match`** `p` **`with`** `S ((S n) as n') → ...`, et d'utiliser la variable `n'`.

7.3.2 Compteur d'itérations

Dans le cas où l'on veut utiliser dans Coq une fonction qui ne termine pas, ou dont on ne sait pas prouver la terminaison, une technique consiste à modifier cette fonction en lui ajoutant un argument de type `nat` qui décroit à chaque appel de la fonction. La fonction a alors la forme suivante :

```
Fixpoint f (p : nat) ... : option ... := match p with
  | 0 ⇒ None
  | S p ⇒ ...
end.
```

Dans tous les appels récursifs, le premier argument sera toujours `p`. Ce paramètre mesure la profondeur des appels récursifs. Ainsi, lorsque la fonction sera appelée, elle renverra un résultat si la profondeur des appels récursifs nécessaire aux calculs ne dépasse pas la valeur de `p`, sinon la fonction renvoie `None`. Ce procédé est par exemple utilisé dans le développement de CompCert [Blazy and Leroy 2009].

Pour compiler des fonctions en utilisant un compteur, il faut les modifier de la même manière que pour les spécifications non complètes de la section 7.2.1.2 : changer le type de retour des fonctions et adapter les filtrages et les valeurs de retour. De plus, il faut aussi ajouter un filtrage sur l'argument décroissant au début de la fonction. Pour l'instant, l'utilisation d'un tel compteur est le seul moyen d'extraire des fonctions vers Coq avec notre outil si ces dernières ne suivent pas un schéma de récurrence structurelle. Si le support de la récursion bien fondée est ajouté, ce procédé sera toujours utile pour les fonctions qui ne terminent pas systématiquement. C'est notamment le cas de l'interprète pour un langage impératif tel que celui présenté dans la figure 4.2.

7.4 Fonctions partielles et dépendances

Nous avons vu dans les sections 7.2.1.2 et 7.3.2 que lors de l'extraction de certaines fonctions, leur type de retour peut changer. Nommons "option" et "compteur" les deux modifications que peuvent subir les fonctions extraites. L'objectif de cette section est d'étudier les interactions qui existent entre ces deux types de modification, en particulier lorsque plusieurs fonctions qui dépendent les unes des autres sont extraites.

7.4.1 Extraction en mode partiel

Une fonction qui subit la modification "compteur" n'a plus à subir la modification "option", car la première modification réalise déjà tous les changements que fait la première. Dans le cas d'une extraction en mode partiel, une fonction subit la modification "option" si

elle est issue d'une spécification incomplète. La modification "compteur", faite à la demande de l'utilisateur, peut être effectuée sur une fonction issue d'une spécification complète comme incomplète.

7.4.2 Extraction en mode complet

Dans le cas d'une extraction en mode complet, la modification "option", seule, n'a pas de sens. En effet, la fonction générée renvoie alors `true` ou `false`, le type `option` n'est donc pas nécessaire. En revanche, la modification "compteur", elle, peut être demandée par l'utilisateur (si la fonction ne termine pas par exemple). Le type de retour de la fonction devient alors `option bool`.

7.4.3 Dépendances

Lorsqu'une relation inductive d utilise une autre relation inductive d' dans sa définition, on dit que d dépend de d'. Si la fonction $f_{d'}$ extraite à partir de d' a subi la modification "option", cela peut avoir une influence sur la fonction f_d extraite à partir de d. En effet, la fonction $f_{d'}$ apparaît dans le corps de la fonction d, et si elle renvoie `None`, la fonction f_d (si elle est extraite avec un mode partiel) devra aussi renvoyer `None`. On est donc contraint d'appliquer la modification "option" à la fonction f_d.

D'une manière générale, l'influence de $f_{d'}$ sur f_d varie suivant les critères suivants :

— Le mode d'extraction de d est-il complet ?

— Le mode d'extraction de d' est-il complet ?

Suivant ces deux critères, les modifications "option" et "compteur" vont se propager de $f_{d'}$ à f_d ou non. Dans le tableau 7.2, chaque colonne est une combinaison des deux paramètres précédents. Sur la première ligne, on suppose que $f_{d'}$ a subi la modification "option", et une croix est présente lorsque cela force la même modification pour f_d. La deuxième ligne est similaire pour la modification "compteur".

Cette analyse a lieu lors de la transformation des Reltree en fonctions MFL.

Mode d/Mode d'	Comp./Comp.	Comp./Part.	Part./Comp.	Part./Part.
Option	-[1]	-[2]	-[1]	X
Compteur	X	X	X	X

[1] La modification "option" n'a pas de sens dans le cas d'une fonction complète.
[2] `None` devient `false` dans f_d.

TABLE 7.2 – Influence des dépendances sur les modificateurs de fonctions extraites

7.5 Implantation et résultats

La compilation vers **`Fixpoint`** est plus compliquée que la compilation vers OCaml, car elle nécessite de vérifier la terminaison et de compiler les filtrages des fonctions MFL vers des filtrages plus élémentaires. Pour l'instant, toutes les fonctionnalités disponibles pour l'extraction vers OCaml ne le sont pas encore pour Coq.

Ainsi, le prédicat d'égalité n'est autorisé dans la définition d'une relation inductive dont on veut extraire un fonction Coq que s'il est utilisé avec les modes $\{1\}$ et $\{2\}$. Avec le mode $\{1, 2\}$, il faudrait disposer du lemme de décidabilité et utiliser ce dernier dans la fonction extraite. Pour certains types prédéfinis, ce lemme existe, comme `eq_nat_dec` par exemple pour le type `nat`. Cependant, pour d'autres types, nous ne disposons pas d'un tel lemme. Deux solutions sont envisagées. La première consiste à générer le lemme d'égalité décidable pour chaque type rencontré dans une égalité. La seconde, à demander à l'utilisateur de fournir un tel lemme. La compilation serait alors stoppée, avec un message d'erreur indiquant la liste des égalités à fournir afin que la compilation réussisse.

La prise en compte des fonctions dans les entrées des prémisses (ou sorties des conclusions), des connecteurs logiques, l'ordonnancement des constructeurs, le traitement des fonctions mutuellement récursives (dans la limite des fonctions récursives structurelles), toutes ces extensions sont mises en œuvre. Certaines limitations sont néanmoins présentes, principalement à cause d'une implantation encore partielle. L'extraction vers OCaml bénéficie de l'extraction native de Coq dont l'implantation est plus mature, et rencontre moins de problèmes à ce niveau.

Les fonctions extraites vers Coq étant utilisables comme n'importe quelle autre fonction, il est possible de les extraire vers OCaml avec l'extraction native de Coq. De même, on peut utiliser une fonction f extraite à partir d'une relation inductive dans une autre fonction g

écrite en Coq par l'utilisateur puis d'extraire g. La fonction f sera alors extraite en tant que dépendance de g. Cela constitue une solution à la limite de l'extraction vers OCaml décrite dans la section 6.2.2.

7.6 Génération automatique des preuves

La génération automatique des preuves est un point central dans le cas de l'extraction vers Coq. En effet, la motivation première de ce travail reste d'offrir à l'utilisateur la possibilité de manipuler à la fois les versions relationnelle et fonctionnelle d'une spécification sans avoir à écrire deux fois le code. Pour que l'utilisateur prouve des propriétés sur la relation ou la fonction, il est primordial qu'une preuve d'équivalence existe entre les deux. Cette preuve doit idéalement être réalisée automatiquement car il est peu agréable de réaliser des preuves sur un code généré. La preuve d'équivalence comporte deux parties : la preuve de correction de la fonction qui assure que tous les résultats renvoyés par cette dernières sont valides pour la relation inductive, et la preuve de complétude de la fonction qui assure que pour toute valeur définie dans la relation inductive, la fonction extraite renvoie bien une valeur.

Nous commençons par présenter le langage de tactiques que nous utilisons pour générer ces preuves dans la section 7.6.1. Ensuite, nous présentons comment produire le schéma d'induction qui constitue la base du raisonnement mis en œuvre dans les preuves dans la section 7.6.2. La preuve d'équivalence de la fonction extraite est scindée en deux parties : une preuve de correction et de complétude. Nous décrirons d'abord le mécanisme général de génération de preuves dans la section 7.6.3, puis nous aborderons la génération des preuves de correction et de complétude dans les sections 7.6.4 et 7.6.5.

7.6.1 Tactiques de preuve de Coq

Nous donnons tout d'abord un aperçu du rôle des tactiques dans la section 7.6.1.1, puis nous abordons les tactiques proprement dites dans la section 7.6.1.2.

7.6.1.1 Le rôle des tactiques

En logique, une règle de déduction est le lien entre une formule que l'on appelle conclusion et un ensemble de formules que l'on appelle prémisses. Une règle de déduction peut être lue de deux manières différentes. La plus naturelle est la suivante : si l'on sait que les prémisses sont vraies alors la conclusion est vraie. Mais on peut également dire que pour prouver la conclusion, il suffit de prouver chacune des prémisses. Une tactique permet de faire le lien entre un but à prouver et un ensemble de nouveaux buts généralement plus simples.

Coq repose sur la correspondance de Curry-Howard. L'interface de Coq, en mode preuve interactive, permet de visualiser le fichier source Coq, mais aussi les buts à prouver. Un but est constitué d'une proposition P que l'on cherche à prouver, et d'un contexte qui contient un ensemble d'hypothèses $\{H_1, \ldots, H_n\}$ que l'on peut utiliser pour prouver la proposition P [Bertot and Castéran 2004]. Une preuve pour le but est un terme t de type P dans un contexte de typage $(h_1 : H_1); \ldots; (h_n : H_n)$. Initialement, le but à prouver est l'énoncé du théorème lui-même. Ensuite, l'utilisateur applique des tactiques pour manipuler le but à prouver. Une tactique peut être définie comme une fonction qui à un but associe plusieurs sous-buts. L'objectif, après plusieurs modifications du but (par l'application de tactiques) est d'obtenir des axiomes. Pour que la preuve soit correcte, les tactiques doivent permettre de reconstruire le but à partir des sous-buts, c'est-à-dire de trouver une preuve pour P à partir des preuves de chacun des sous-buts.

Une tactique peut être décrite comme une fonction ML qui a pour interface :

```
type tactic = goal → (goal list * validation)
and validation = term list → term
```

La "validation" permet de construire le terme preuve correspondant à la tactique. En réalité, l'interface des tactiques de Coq a évolué pour devenir plus complexe [Spiwack 2010], mais nous conservons ici cette version simplifiée car elle est suffisante pour l'utilisation que nous faisons des tactiques.

7.6.1.2 Tactiques primitives de Coq

Certaines tactiques de Coq sont très élaborées et permettent de réaliser simultanément plusieurs manipulations sur le but à prouver. La tactique **`simpl`**, par exemple permet de réaliser des calculs dans le but à prouver. La tactique **`auto`** est utile pour terminer les preuves lorsque ce qui reste à prouver est "trivial". Cependant, nous nous plaçons dans un contexte de génération automatique de preuves, alors que ces tactiques ont été conçues pour l'utilisateur qui réalise des preuves de manière interactive. Lorsque l'on génère un script de preuve automatiquement, toute la preuve est générée avant son exécution. On doit donc pouvoir contrôler avec précision l'effet que vont avoir les tactiques sur le déroulement de la preuve. Étant donné que la documentation des tactiques de Coq [development team 2004] ne donne pas de sémantique précise des tactiques, nous nous contentons d'utiliser les tactiques dont on peut facilement deviner l'effet. Parmi ces tactiques, celle que nous utilisons le plus est la tactique **`replace`** qui permet de remplacer un terme donné par un autre terme, dans la conclusion ou dans une hypothèse du but à prouver. En utilisant cette tactique, on connaît exactement le but que l'on obtient après son application. Son utilisation demande des efforts supplémentaires car il faut écrire précisément sur quels termes elle s'applique. D'autres tactiques comme **`subst`**, permettent automatiquement de substituer une partie d'une égalité par l'autre partie. Cependant, le résultat de cette application est incertain.

Il existe deux niveaux dans les tactiques Coq. Le premier concerne les tactiques utilisées directement par l'utilisateur lorsque ce dernier réalise des preuves interactives. Le deuxième niveau est une interface OCaml que l'on peut utiliser dans les plugins. Les tactiques OCaml ne sont pas identiques aux tactiques proposées à l'utilisateur, et manipulent directement des termes Coq en syntaxe abstraite. Nous générons systématiquement les preuves dans les deux niveaux : une preuve avec les tactiques OCaml qui est effectivement utilisée, et un script de preuve sous forme de chaîne de caractères, équivalent à la première preuve (parfois les tactiques peuvent différer). Ce script peut être affiché et utilisé à des fins de vérification ou de documentation. Dans la suite, nous utilisons ce script pour présenter les preuves.

7.6.2 Schémas d'induction pour les fonctions

Comme nous l'avons précisé dans la subsection 4.2.1, Coq offre la possibilité de raisonner par induction sur les fonctions. Pour cela, un schéma d'induction doit être créé pour la fonction, de même qu'il est généré pour les inductifs Coq (par inductif, nous entendons les types de données et les relations inductives, autrement dit tout ce qui est défini à l'aide du mot clé **`Inductive`**). Si le schéma d'induction pour les inductifs est généré systématiquement lors de la définition de ce dernier, celui associé aux fonctions est généré lors de l'invocation de la commande suivante :

```
Functional Scheme <scheme_name> := Induction for <func_name> Sort Prop.
```

Le schéma d'induction est créé en suivant précisément les chemins d'exécution de la fonction. On peut donc l'utiliser pour produire les preuves automatiques car on peut connaître sa forme en utilisant les chemins d'exécution de la fonction extraite dans le langage CIC.

7.6.3 Principe général de la génération des preuves

Le schéma général de génération des preuves est donné dans la figure 7.3. Les preuves sont toujours générées à partir des chemins d'exécution de la fonction, sur lesquels on a ajouté des annotations. Ces preuves utilisent le schéma d'induction généré par Coq.

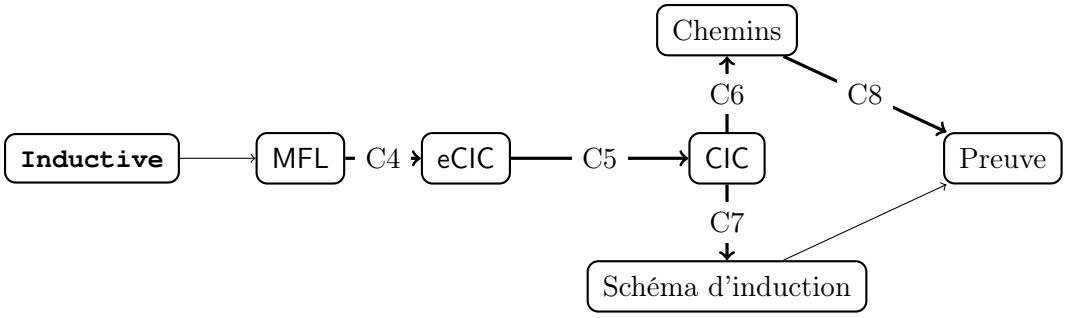


FIGURE 7.3 – Schéma de compilation des preuves

Le code des fonctions extraites est généré à partir d'une relation inductive. L'idée sous-jacente à la génération de preuve est d'établir une correspondance entre la fonction et la définition inductive. Pour cela, on raisonne par induction et la preuve se fait avec un cas par chemin d'exécution de la fonction ou du Reltree. De plus, on sait, d'après l'algorithme

de génération des fonctions, qu'à chaque chemin d'exécution correspond un constructeur de la définition inductive. On peut donc établir, pour chaque sous-but à prouver, un lien entre ce sous-but, un chemin d'exécution de la fonction, un chemin dans le Reltree et un constructeur de la relation inductive. Il peut y avoir plusieurs chemins d'exécutions rattachés à un même constructeur. Les étapes de la preuve sont les suivantes :

— appliquer le schéma d'induction au but à prouver,

— pour chaque sous-but G_i obtenu ($i \in 1..n$), on recherche le chemin d'exécution b_i correspondant, et le constructeur C_i qui a servi à produire le code contenu dans b_i, puis on prouve G_i en appliquant des tactiques qui vont transformer G_i jusqu'à obtenir C_i. Il suffit alors d'appliquer C_i pour terminer la preuve de G_i.

Les annotations placées sur le chemin d'exécution permettent de connaître à partir de quel constructeur et de quelle prémisse a été généré chaque élément du code de la fonction. Elles sont indispensables pour faire le lien entre le constructeur C_i et le sous-but G_i.

Dans l'implantation, les annotations suivent le processus d'extraction. Elles sont dans un premier temps placées sur les Reltree, puis sur les fonctions MFL et CIC. Les chemins d'exécutions annotés sont ensuite créés à partir de ces fonctions.

Un chemin d'exécution annoté est défini de la manière suivante :

Définition 7 (Chemin d'exécution annoté) *Un chemin d'exécution annoté est défini ainsi :*
$b ::= t \mid let_l\ x\ :=\ t\ in\ b$
$\quad \mid\ match\ t\ with\ c\ x_1\ \ldots\ x_p\ \Rightarrow_l\ b$
où t est un terme et l une annotation qui est soit un ensemble de noms de constructeurs $\{C_1,\ \ldots,\ C_n\}$, soit un ensemble de positions de prémisses $\{(C_1, i_1),\ \ldots,\ (C_n, i_n)\}$, où (C_i, i_k) dénote la i_k^{eme} prémisse du constructeur C_i.

7.6.4 Preuves de correction

Les preuves que nous générons utilisent la version OCaml, internes à Coq, des tactiques. Elles sont donc difficilement représentables. Nous présentons donc dans cette section des scripts équivalents aux preuves que nous générons, en utilisant la syntaxe du niveau utili-

sateur de Coq. En conséquence, les preuves présentées peuvent être rejouées dans Coq.

D'autres méthodes ont été envisagées pour générer les preuves. Par exemple, nous aurions pu générer un terme ayant le type qui correspond au but à prouver. Néanmoins, un tel terme est relativement complexe, et la génération d'un script de preuve avec les tactiques s'est révélé plus simple. Un terme peut être affiché une fois le script de preuve joué dans Coq. Ce terme est calculé par les tactiques. Une autre technique possible consiste à créer une tactique spécifique qui permet de prouver le lemme de correction en utilisant d'autres tactiques. Il est alors possible d'utiliser le langage $\mathcal{L}_{tac}$ [Delahaye 2000] plutôt que d'assembler les tactiques avec OCaml. Là encore, la génération du script de preuve s'est révélée plus simple.

Afin de prouver la correction de la fonction extraite par rapport à la spécification de la relation inductive, nous prouvons que tous les résultats qui sont renvoyés par la fonction extraite sont conformes à la spécification. Nous prouvons pour cela le lemme suivant :

$$\forall p_1,\ \ldots,\ p_n, f_d\ p_1\ \ldots\ p_{n-1} = p_n \Rightarrow d\ p_1\ \ldots\ p_n$$

où f_d est le nom de la fonction extraite à partir de la relation inductive d avec le mode $\{1,\ \ldots,\ n-1\}$. Nous supposons dans un premier temps que la fonction est extraite avec un mode partiel, et qu'un seul des arguments est choisi comme sortie. Le mode doit donc contenir tous les arguments de la relation sauf un. La notation ci-dessus suppose également que la sortie est le dernier argument. Ce dernier point n'est qu'une simplification de la notation, mais nullement une restriction de l'algorithme de génération de preuve. Nous supposons de plus que la fonction extraite est une fonction complète et qu'elle suit un schéma d'induction structurelle (pas d'utilisation du compteur).

La première étape de la preuve, qui consiste à appliquer le schéma d'induction f_d_ind lié à la fonction f_d, préalablement généré, en utilisant les tactiques suivantes :

```
intros until 0.   (* introduction des variables *)
intro H.          (* introduction de f_d p_1 ... p_n-1 = p_n *)
subst p_n.        (* remplace p_n par (f_d p_1 ... p_n-1) dans le but *)
apply fd_ind.     (* applique le schema d'induction *)
```

Il y a autant de sous-buts que de chemins d'exécution dans la fonction f_d. Appelons

n le nombre de sous-buts. Une fois le schéma inductif appliqué, on obtient n sous-but à prouver ($G_i, i \in 1..n$). Pour tout i, on note b_i et C_i respectivement le chemin d'exécution et le constructeur (de d) associés à G_i.

Chaque sous-but G_i a la forme suivante :

$$\forall \overrightarrow{v}, (\forall \overrightarrow{v_1}, \overrightarrow{a_1} \to H_1) \to \ldots \to (\forall \overrightarrow{v_k}, \overrightarrow{a_k} \to H_k) \to d\ t_1 \ldots t_p$$

où $\overrightarrow{v}$, $\overrightarrow{v_k}$ (pour $k \in 1..j$) sont des listes de variables, j le nombre de prémisses associées au constructeur C_i. Le vecteur $\overrightarrow{a_k}$ est une liste d'égalités et de constructions **`let in`**. Les égalités correspondent aux filtrages présents dans b_i et les **`let in`** aux **`let in`** présents dans b_i. Chaque groupe "$\forall \overrightarrow{v_k}, \overrightarrow{a_k} \to H_k$" (notons le p_k) correspond à une prémisse de C_i. Rappelons que le sous but G_i est un cas du raisonnement par induction sur les chemins d'exécution de la fonction. Or, chaque portion de code du chemin d'exécution b_i a été généré soit à partir d'une prémisse de C_i, soit à partir de sa conclusion. Chaque groupe p_k est donc issu du code qui a été généré à partir d'une prémisse de C_i. Notons cette prémisse P_k. Le terme H_k, qui dépend de $P_k = d_k\ \ldots$, est :

— soit $d\ \ldots\ (f_d\ \ldots)\ \ldots$ si $d_k = d$.

— soit $f_{d_k}\ \ldots = \ldots$ ou $let\ \ldots = f_{d_k}\ \ldots in$ si $d_k \neq d$.

On ne tient pas compte du nom des variables pendant la génération de la preuve. Elles seront simplement introduites à l'aide de la tactique **`intros`**, et un nouveau nom leur sera alors attribué.

Les $\overrightarrow{a_k}$ sont issus de la compilation des filtrages vers des filtrages de bas niveau (de eCIC vers CIC). Pour retrouver la forme de la prémisse ou de la conclusion initiale, on doit donc effectuer la transformation inverse. Pour cela, on va remplacer la partie gauche des égalités (ou la variable des **`let in`** dans le code généré) par la partie droite (ou la valeur dans les **`let in`**), dans tout le sous-but. En effet, la partie gauche du sous-but est toujours une variable intermédiaire utilisée pour compiler le filtrage et la partie droite est le terme qui correspondait à cette variable dans la spécification initiale. Cependant, toutes les égalités ne doivent pas être utilisées. En effet, il arrive parfois lors de la fusion de constructeurs que le remplacement ne concerne que l'un des constructeurs.

Le script de preuve pour chaque G_i est généré à partir de b_i de la manière suivante :

1. On identifie dans G_i les différentes parties : $\overrightarrow{v}$, $\overrightarrow{v_k}$, $\overrightarrow{a_k}$, H_k.
2. On introduit l'ensemble de ces constructions en nommant les hypothèses.
3. On effectue les transformations nécessaires pour chaque a_{kh} (remplacement de termes en utilisant des égalités), en utilisant b_i.
4. On "remonte" les hypothèses qui correspondent aux H_k dans le but à prouver.
5. Et on applique le constructeur C_i.

Détaillons chacune de ces étapes, en précisant comment elles sont réalisées :

1. La première étape consiste à attribuer un nom à chaque variable et chaque prémisse de G_i. Chacun de ces éléments va ensuite devenir une hypothèse dans le but à prouver lors de l'application de la tactique [1] **`intros`**.
2. La seconde étape consiste à appliquer la tactique **`intros`** avec les noms choisis à la première étape.
3. On note $[\![b_i]\!]$ la génération de preuve pour un chemin d'exécution b_i. Cette étape est alors réalisée selon les deux algorithmes suivants :

   — Pour un **`let in`** : $[\![let_l\ x = t\ in\ b]\!] \triangleq \begin{cases} assert\ (xEQ\ :\ x\ =\ t). \\ auto. \\ replace\ (x)\ with\ (t)\ in\ *. \\ [\![b]\!] \end{cases}$

   Le but est de remplacer `x` par `t` partout là où cela est possible. Pour utiliser la tactique **`replace`** il faut que l'égalité `x = t` soit présente dans les hypothèses. Pour cela on utilise la tactique `assert`, qui introduit un nouveau but à prouver. Ce but supplémentaire est prouvé avec la tactique **`auto`**.

   — Pour un **`match`** : $[\![match\ t\ with\ c\ x_1\ \ldots\ x_p \Rightarrow_l b]\!] \triangleq$

   $\begin{cases} \text{si } C_i \text{ apparaît dans } l \text{ alors} \\ \quad replace\ (t)\ with\ (c\ x_1\ \ldots\ x_p)\ in\ *. \\ \quad [\![b]\!] \\ \text{sinon } [\![b]\!] \end{cases}$

   C_i est le constructeur de la relation inductive associé à G_i. Dans le cas où C_i n'apparaît pas dans l'annotation l, le code du filtrage ne concerne pas le constructeur C_i. Il s'agit de code généré pour un autre constructeur de la relation inductive.

1. Le mot "tactique" désigne ici une tactique OCaml.

4. La quatrième étape est réalisée en appliquant la tactique suivante :

   ```
   revert HREC1 ... HRECj
   ```

   où `HRECx` est le nom donné à l'étape 1 aux prémisses qui sont l'application d'une relation inductive d_i (différente de d) avec ses arguments. La tactique **revert** permet de faire apparaître une hypothèse dans le but à prouver.

5. Enfin, la cinquième étape consiste simplement à appliquer le constructeur C_i :

   ```
   apply Ci; assumption.
   ```

Une fois la preuve de tous les sous-buts générée avec la méthode précédente, le script de preuve est exécuté et le lemme prouvé est sauvegardé dans l'environnement global de Coq.

La figure 7.5 montre par exemple la preuve de correction pour la fonction présentée dans la figure 7.4. Le nom des variables et des hypothèses, ainsi que l'ordre de certaines tactiques diffèrent du résultat qui serait obtenu avec l'algorithme présenté mais il ne s'agit que de choix d'implantation. Les deux algorithmes sont néanmoins équivalents et les preuves similaires.

```
Fixpoint add12 (p1 p2 : nat) {struct p2} : nat :=
  let fix_5 := p1 in
  match p2 with
  | 0 ⇒ fix_5
  | S fix_6 ⇒ let fix_7 := add12 fix_5 fix_6 in S fix_7
  end.
```

FIGURE 7.4 – Fonction add12 générée automatiquement

Pour l'instant, l'algorithme de génération de preuves ne fonctionne que dans le cas des fonctions complètes et sans compteur (sans type option) et pour les modes avec une unique sortie. Le procédé précédent semble néanmoins généralisable.

Pour une fonction partielle, le lemme à prouver est légèrement différent, puisqu'il doit tenir compte du type option :

$$\forall p_1,\ \ldots,\ p_n, f_d\ p_1\ \ldots\ p_{n-1} = \mathit{Some}\ p_n \Rightarrow d\ p_1\ \ldots\ p_n$$

```
Lemma add12_correct :
  forall p1 p2 p3, add12 p1 p2 = p3 → add p1 p2 p3.
Proof.


intros until 0.
intro H.
subst po.
apply add12_ind.

(* 1er chemin *)
(* etapes 1 et 2 *)
intros na_10 na_9 fix_5 HCC_8.
(* etape 3 *)
assert (fix_5EQ : fix_5 = na_10).
auto.
replace (fix_5) with (na_10) in *.
replace (na_9) with (0) in *.
(* etape 5 *)
apply add0; try assumption.

(* 2eme chemin *)
(* etapes 1 et 2 *)
intros na_15 na_14 fix_5 na_13 HCC_11 HREC_12 fix_7.
(* etape 3 *)
assert (fix_5EQ : fix_5 = na_15).
auto.
assert (fix_7EQ : fix_7 = add12 fix_5 na_13).
auto.
replace (fix_5) with (na_15) in *.
replace (na_14) with (S na_13) in *.
replace (fix_7) with (add12 na_15 na_13) in *.
(* etape 4 *)
revert HREC_12.
(* etape 5 *)
apply addS; try assumption.
Qed.
```

FIGURE 7.5 – Script de preuve généré automatiquement pour la fonction add12

Pour une fonction extraite avec le mode complet, le lemme est également différent :

$$\forall p_1,\ \ldots,\ p_n, f_d\ p_1\ \ldots\ p_n = true \Rightarrow d\ p_1\ \ldots\ p_n$$

Et pour une fonction avec compteur, le lemme a la forme suivante :

$$\forall p_1,\ \ldots,\ p_n, \forall c, f_d\ c\ p_1\ \ldots\ p_{n-1} = Some\ p_n \Rightarrow d\ p_1\ \ldots\ p_n$$

7.6.5 Preuves de complétude

Nous ne réalisons pas encore automatiquement les preuves de complétude des fonctions extraites.

Dans le cas d'une fonction complète, le lemme de complétude s'écrit :

$\forall p_1,\ \ldots,\ p_n, d\ p_1\ \ldots\ p_n \Rightarrow f_d\ p_1\ \ldots\ p_{n-1} = p_n$

Dans le cas d'une fonction partielle, le lemme à prouver est le suivant :

$\forall p_1,\ \ldots,\ p_n, d\ p_1\ \ldots\ p_n \Rightarrow f_d\ p_1\ \ldots\ p_{n-1} = Some\ p_n$

Dans le cas de l'extraction avec un mode complet, le fait de prouver ce lemme est indispensable, car sinon il est suffisant d'extraire la fonction qui renvoie toujours `false` et qui peut être trivialement prouvée correcte. Ce lemme s'écrit :

$\forall p_1,\ \ldots,\ p_n, d\ p_1\ \ldots\ p_n \Rightarrow f_d\ p_1\ \ldots\ p_n = true$

Nous avons fait les preuves manuellement pour un certain nombre d'exemples, par induction sur l'hypothèse inductive, mais aucun algorithme général n'a été implanté pour l'instant afin de prouver ces lemmes de complétude.

Chapitre 8

Extraction dans l'environnement Focalize

Nous proposons dans ce chapitre une autre implantation du "framework" décrit dans le chapitre 5, dans l'environnement Focalize. Cette implantation partage le même code que celle de Coq en ce qui concerne la transformation des spécifications en code MFL. Cela démontre la pertinence des interfaces que nous avons choisies, car elles permettent de s'adapter facilement à un autre environnement de preuve. Pourtant, Focalize ne possède pas de construction pour exprimer nativement les relations inductives. Nous commençons ce chapitre avec une présentation de Focalize (dans la section 8.1), mettant en avant les spécificités de cet environnement centré autour du concept d'héritage des spécifications et des preuves. Ensuite, nous détaillons comment nous avons émulé les relations inductives à l'aide de propriétés exprimées dans la logique du premier ordre dans la section 8.2, puis nous abordons l'implantation dans la section 8.3 et la génération des preuves dans la section 8.4. Enfin, nous terminons par une discussion sur les perspectives qu'ouvrent le mariage de notre approche concernant l'extraction et le formalisme de Focalize dans la section 8.5.

8.1 Présentation de Focalize

8.1.1 L'environnement Focalize

Le projet Focalize (initialement projet Foc, puis Focal) a débuté en 1997 sous l'impulsion de T. Hardin et R. Rioboo. Il s'agit d'un environnement dans lequel il est possible de

construire des applications pas à pas, partant de spécifications abstraites, et allant vers des implantations concrètes par des étapes de raffinement successives. Il permet également d'énoncer des propriétés et de prouver que les programmes respectent leurs spécifications. Ces raffinements concernent aussi bien le code exécutable que la preuve de propriétés. Le compilateur produit du code OCaml pour l'exécution, du code Coq pour la certification formelle, mais aussi du code XML pour la documentation. Zenon [Bonichon et al. 2007], un outil de déduction automatique au premier ordre, permet d'automatiser une partie des preuves. Le processus de compilation au sein de l'atelier Focalize est réalisé en plusieurs étapes.Tout d'abord, le compilateur Focalize génère pour chaque fichier source un fichier OCaml et un squelette de fichier Coq. Le squelette de fichier Coq contient des preuves à trous qui sont ensuite complétées par Zenon. Coq compile ensuite le fichier complété, et enfin le compilateur OCaml compile le fichier exécutable.

8.1.2 Spécification et implantation dans Focalize

La brique de base du langage Focalize est la notion d'"espèce", qui partage un certain nombre de points commun avec la notion de classe dans le paradigme objet. Une espèce peut être globalement vue comme une liste d'attributs de trois sortes :

- le type support, appelé "représentation", qui est le type des entités qui sont manipulées par les fonctions de l'espèce ; la représentation peut être soit abstraite soit concrète ;
- les fonctions, qui dénotent les opérations permises sur les entités de la représentation ; les fonctions peuvent être soit des définitions (quand un corps est fourni), soit des déclarations (quand seul le type est donné) ;
- les propriétés, qui doivent être vérifiées par toute implantation de l'espèce ; les propriétés peuvent être soit simplement des énoncés, soit des théorèmes (quand la preuve est également fournie).

La syntaxe d'une espèce est la suivante :

```
species <name> =
  representation [= <type>];        (* representation *)
  signature <name> : <type>;        (* declaration *)
  let <name> = <body>;              (* definition *)
  property <name> : <proposition>; (* propriete *)
```

```
theorem <name> : <proposition>
  proof = <proof>;                (* theoreme *)
proof of <name> = <proof>;        (* preuve d'une propriete *)
```

où <name> est un nom, <type> une expression de type, <body> un corps de fonction (dans une syntaxe proche du noyau purement fonctionnel de OCaml), <proposition> une proposition logique (du premier ordre) et <proof> une preuve (exprimée au moyen d'un langage de preuve déclaratif ou d'un script Coq). Dans le langage des types, le mot-clé "Self" fait référence au type de la représentation et peut être utilisée partout dans une espèce sauf dans la représentation elle-même.

Les espèces peuvent être combinées en utilisant l'héritage (possiblement multiple), qui fourni un mécanisme similaire à celui qu'on trouve dans les langages orientés objet. Il est possible de définir dans l'espèce "fille" des fonctions qui étaient précédemment seulement déclarées ou de prouver des propriétés qui n'avaient pas de preuve. Il est aussi possible de redéfinir des fonctions précédemment définies ou de prouver à nouveau des propriétés déjà prouvées. Cependant, la représentation ne peut pas être redéfinie et les fonctions ainsi que les propriétés doivent garder leurs types respectifs tout au long du chemin d'héritage. En pratique, l'héritage peut être utilisé pour étendre une espèce en y ajoutant de nouvelles fonctionnalités, et/ou pour raffiner une espèce en implantant sa spécification.

La redéfinition de fonctions invalide les preuves qui dépendent de la définition de la fonction. Ainsi, une propriété prouvée dans une espèce e en utilisant la définition d'un fonction f devra être prouvée de nouveau dans une espèce fille si cette dernière redéfinit la fonction f.

Lorsque toutes les déclarations d'une espèce sont définies, et que toutes les propriétés sont prouvées, on dit que l'espèce est une espèce complète. De plus, il est possible de créer, à partir d'une espèce complète, une collection qui cache la représentation abstraite de l'espèce. La collection est assimilable à un nouveau type et il n'est plus possible d'hériter d'une collection.

En plus de l'héritage, une seconde façon de combiner les espèces consiste à utiliser la paramétrisation. Les espèces peuvent être paramétrées soit par des collections soit par des entités provenant de collections. Si le paramètre est une collection, l'espèce paramétrée

a seulement accès à l'interface de cette collection, c'est-à-dire seulement sa représentation abstraite, ses déclarations et ses propriétés. Ces deux mécanismes de combinaison d'espèces complètent la syntaxe précédente comme suit :

```
species <name> (<name> is <name>[(<pars>)], <name> in <name>,
                                                          ...) =
  inherit <name>, <name> (<pars>), ...;
end;;
```

où *<pars>* est une liste de *<name>*, qui dénote les noms utilisés comme paramètres effectifs. Quand le paramètre est une collection, le mot-clé "is" est utilisé. Quand c'est une entité, le mot-clé "in" est utilisé.

Une collection peut être vue comme la version finale d'une espèce, dans laquelle chaque attribut est concret : la représentation est concrète, les fonctions sont définies et les propriétés sont prouvées. Si l'espèce implantée est paramétrée, la collection doit aussi fournir des implantations pour ces paramètres : soit une collection s'il s'agit d'un paramètre de collection, soit une entité s'il s'agit d'un paramètre d'entité. La syntaxe d'une collection est la suivante :

```
collection <name> = implement <name> (<pars>) end;;
```

8.1.3 Preuves dans Focalize

Mener un développement Focalize requiert de prouver que les propriétés énoncées sont valides. Il est donc nécessaire de les démontrer. Pour ce faire, un outil de preuve automatique au premier ordre (basé sur la méthode des tableaux), appelé Zenon [Bonichon et al. 2007], aide l'utilisateur à construire les preuves. Le langage de preuve est un langage déclaratif qui permet d'indiquer à Zenon les étapes de la preuve en précisant quelles définitions et quelles propriétés utiliser. Pour faire ces preuves, il est également possible de donner directement une preuve Coq, sous forme d'un script de preuve. Cette option peut s'avérer utile lorsqu'une preuve fait appel à certains aspects logiques que Zenon ne sait pas gérer. En revanche, cela demande une bonne connaissance des mécanismes de compilation de Focalize car la preuve se fait sur les éléments qui sont produits par le compilateur. Les preuves Coq sont introduites comme suit :

```
theorem <name> : <prop>
```

```
proof =
  coq proof
  definition of <name>, ...
  {* <coq> *};
```

où *<name>* est le nom d'une définition utilisée dans la preuve Coq, et *<coq>* une preuve Coq.

8.2 Relations dans Focalize

Contrairement à Coq, Focalize ne permet pas de définir des relations définies de manière inductive. Cependant, même si les relations inductives ne font pas partie du langage de spécification de Focalize, certaines constructions de ce langage permettent d'exprimer des spécifications qui expriment des relations. Ainsi, en utilisant une déclaration de fonction, et des propriétés du premier ordre, il est possible d'écrire la spécification de la figure 8.1. Cette espèce n'est pas sans rappeler la spécification de la relation inductive `add` en Coq. Néanmoins, l'espèce `AddSpecif` ne spécifie pas, au contraire de Coq via sa construction **`Inductive`**, la plus petite relation vérifiant les propriétés énoncées.

```
type nat = | Zero | Succ (nat);;

species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add (n, Zero, n);
  property addS : all n m p : nat, add (n, m, p) →
    add (n, Succ (m), Succ (p));
end;;
```

FIGURE 8.1 – Spécification de l'addition dans une espèce Focalize

En effet, même si l'espèce de la figure 8.1 ressemble à une relation inductive, elle n'en est pas une. Tout d'abord, rien ne lie syntaxiquement les propriétés entre elles. Les deux propriétés `addZ` et `addS` ne sont rattachées à `add` que parce que `add` apparaît dans ces propriétés. Cela n'est pas suffisant pour affirmer que ce sont ces propriétés qui définissent `add`. Le mécanisme d'héritage de Focalize permet par exemple d'ajouter des propriétés qui peuvent contraindre `add` dans une autre espèce, comme le montre la figure 8.2. Pour cette raison, nous avons fait le choix de laisser à l'utilisateur le soin de spécifier lui-même les

relations inductives, en précisant la liste des propriétés qui définissent cette relation. La liste des propriétés est donnée au moment de l'extraction. Nous pouvons ainsi extraire du code à partir de certaines espèces Focalize.

```
species AddExtension =
  inherits AddSpecif;
  property addWeird : all n : nat, add (n, n, n);
end;;
```

FIGURE 8.2 – Modification d'une spécification en utilisant l'héritage

Pour résumer, nous n'avons pas introduit la notion de relation inductive dans Focalize. Cette notion n'existe que virtuellement, dans le but d'extraire une fonction. Si l'on voit un programme Focalize comme une arborescence d'espèces, il est possible de placer une déclaration de relation et des propriétés (qui caractérisent celle-ci) partout dans cet arbre. L'utilisateur peut alors demander l'extraction d'une relation inductive qu'il définit lui-même en choisissant une déclaration de fonction booléenne et une liste de propriétés, qui doivent simplement être visibles dans l'espèce où est demandée l'extraction. Il doit également préciser les modes d'extraction comme précédemment.

Nous nommons spécification l'ensemble formé par une déclaration de fonction booléenne et un ensemble de propriétés. Pour réaliser une extraction, il faut une spécification et un mode. Afin d'être acceptée dans le processus d'extraction, une spécification doit obéir à certaines contraintes. Tout d'abord, tous les arguments de la fonction déclarée doivent être des types inductifs, et le type de retour doit être `bool`. De plus, les propriétés doivent avoir la même forme que les constructeurs décrits dans la section 5.1. De manière générale, le formalisme de Focalize est plus restrictif que celui de Coq. Certaines limitations imposées dans la section 5.1 le sont déjà dans le cadre de Focalize, comme le premier ordre dans les propriétés.

8.3 Implantation de l'extraction dans Focalize

Contrairement à l'implantation Coq, celle de Focalize n'introduit pas de nouvelle commande. Elle n'introduit pas non plus de nouvelle construction dans le langage, mais s'intègre naturellement dans l'environnement Focalize grâce au système des annotations présent dans

ce dernier. Les annotations sont des commentaires qui disposent d'un en-tête, `Pred` dans notre cas, et qui sont destinées à être traités par un outil spécifique (précisé par l'en-tête). Elles sont d'ores et déjà utilisées par FoCDoc [Maarek and Prevosto 2003], l'outil de production de documentation automatique pour Focalize. Nous utilisons les annotations, placées sur une signature, pour déclencher l'extraction et préciser la relation et les propriétés utilisées. L'extraction fonctionne alors de la manière suivante : l'utilisateur spécifie une relation en donnant son type et en écrivant un certain nombre de propriétés qui décrivent cette relation. Ces dernières, ainsi que la signature qui représente la relation peuvent être disséminées dans plusieurs espèces qui partagent des liens d'héritage. Ensuite, afin d'en extraire une fonction, il écrit la signature de la fonction à extraire (dans le cas du mode complet, cela n'est pas nécessaire car il peut utiliser la relation elle-même) et place une annotation sur cette dernière afin que notre outil remplace cette signature par une fonction définie.

Par exemple, dans la spécification de la figure 8.3, la relation `add` est extraite avec le mode $\{1, 2\}$. Pour cela, on donne la signature de la fonction que l'on veut extraire : `add12`, puis on annote cette signature de manière à préciser de quelle relation la définition doit être extraite, le mode d'extraction et les propriétés à utiliser. La fonction `add12` sera définie automatiquement et utilisable directement dans l'espèce `AddImp`. On remarque que la signature `add12` ne se trouve pas dans la même espèce que la relation `add`. De plus, la notation pour le mode est différente de celle utilisée dans l'implantation pour Coq. Ici, on précise pour chaque argument, dans l'ordre, s'il s'agit d'une entrée (`I`) ou d'une sortie (`O`) pour la fonction extraite.

Dans la figure 8.4, l'extraction est réalisée en mode complet. La signature de la relation à extraire et celle de la fonction extraite partagent alors le même type. Il est alors possible d'utiliser la signature de la relation elle-même comme support de l'extraction. On place pour cela l'annotation directement sur cette dernière.

Pour éviter les confusions, dans le cadre de l'extraction dans Focalize, nous nommons relation la signature qui sert de support à l'extraction (`add` par exemple). En revanche, nous gardons l'appellation signature pour la déclaration de fonction qui est remplacée au moment de l'extraction par une fonction définie (`add12` par exemple). Dans le cas d'une

```
species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add(n, Zero, n);
  property addS : all n m p : nat, add(n, m, p) →
    add(n, Succ(m), Succ(p));
end ;;

species AddImp =
  inherit AddSpecif;
  (** {Pred} add12 : add (I, I, O) with (addZ, addS) *)
  signature add12 : nat → nat → bool;
end ;;
```

FIGURE 8.3 – Extraction de la fonction add12 dans une espèce Focalize

```
species AddFull =
  (** {Pred} add : add (I, I, I) with (addZ, addS) *)
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add(n, Zero, n);
  property addS : all n m p : nat, add(n, m, p) →
    add(n, Succ(m), Succ(p));
end ;;
```

FIGURE 8.4 – Extraction de la fonction add dans une espèce Focalize

extraction en mode complet, ces deux notions sont confondues, la même signature pouvant jouer les deux rôles.

8.4 Preuves de correction

Tout comme pour l'extraction vers Coq, la génération des preuves est une partie importante du travail d'extraction dans Focalize. L'objectif est de montrer que les sorties calculées par la fonction extraite, quelles que soient les entrées, vérifient chacune des propriétés ayant participé à l'extraction. Une version très simple de la génération des preuves a été étudiée dans [Delahaye et al. 2010], mais elle ne fonctionne que dans les cas les plus simples. Lors de ce premier travail, la réalisation des preuves étaient difficiles car Zenon ne gérant pas l'induction, il était impossible de l'utiliser. Les preuves devaient alors être générées sous forme d'un script de preuve, intégré au code Coq généré par le compilateur Focalize. Il fallait alors tenir compte des particularités d'implantation inhérentes au compilateur dans les preuves.

Aujourd'hui, Zenon gère l'induction. La génération de preuves pour ce dernier constitue une perspective intéressante. S'il est possible de se contenter de fournir une preuve déclarative, signalant les propriétés et définitions à utiliser à chaque étape de la preuve, la génération sera peut-être plus aisée que dans le cas de Coq, même si Zenon ne dispose pas d'outil pour gérer l'induction fonctionnelle.

8.5 Discussion sur les liens entre extraction et héritage

L'implantation que nous avons réalisée dans Focalize laisse à l'utilisateur une grande liberté en ce qui concerne l'extraction de fonctions, puisque l'extraction n'est régie que par des annotations et que l'utilisateur peut les compléter comme il le souhaite. De plus, contrairement aux relations inductives de Coq, qui sont définies une fois pour toutes, rien n'empêche dans Focalize d'hériter d'une espèce dans laquelle on a réalisé une extraction, et d'ajouter dans l'espèce fille des propriétés sur la relation. Le code généré par l'extraction a alors peu de chance de vérifier cette nouvelle propriété. En revanche, rien n'empêche l'utilisateur de placer une nouvelle extraction dans l'espèce fille, afin de tenir compte de la nouvelle propriété.

Ainsi, on peut voir l'extraction dans Focalize comme un mécanisme de génération de code automatique, dans une espèce donnée. Là où l'utilisateur utiliserait traditionnellement une déclaration de fonction, il a maintenant la possibilité (pour peu que l'extraction n'échoue pas) de se contenter d'écrire une signature annotée qui sera transformée à la compilation par la fonction extraite. Il pourra alors considérer dans la suite du développement que la fonction est effectivement définie, ouvrant la possibilité de créer une collection.

Chapitre 9

Résultats et performances

Ce dernier chapitre fait le bilan des résultats obtenus durant la thèse au niveau des implantations. Dans un premier temps, nous faisons le point sur les outils et les fonctionnalités qui ont été développées dans la section 9.1. La section 9.2 propose un comparatif de notre outil d'extraction vers OCaml avec deux autres outils : Isabelle/HOL et Twelf.

9.1 Résultats

Le tableau 9.1 dresse un bilan des fonctionnalités présentes dans les différentes implantations. Les différentes optimisations sont décrites dans le chapitre 5.

L'implantation Focalize est distribuée au sein du compilateur depuis la version 0.8 qui peut être téléchargé à l'adresse [http ://focalize.inria.fr/ 2013].

L'implantation Coq est distribuée sous forme d'un plugin qui regroupe l'extraction vers OCaml et l'extraction vers Coq. Ce plugin d'environ 6000 lignes de code fait partie des contributions de Coq et il peut être téléchargé depuis le site officiel de l'outil d'aide à la preuve [Tollitte et al. 2013a]. La dernière version est disponible sur le dépôt qui sert au développement du plugin [Tollitte et al. 2013b]. Une série d'exemples est présente dans l'archive du plugin à des fins de test et d'illustration. Comme le montrent les exemples de ce chapitre, notre plugin peut être utilisé pour extraire du code à partir de spécifications non triviales.

Les exemples présentés jusqu'à présent ont tous été spécifiquement écrits pour illustrer notre approche ou pour tester notre implantation. Même si certains sont suffisamment

	Code OCaml	Code Coq	Preuves	Code Focalize
Mode partiel[1]	✓	✓	✓	✓
Mode partiel[2]	✓	✓		✓
Mode complet	✓	✓		✓
Prédicat égalité	✓	$\sim$ [3]	$\sim$ [3]	✓
Non linéarité	✓			✓
Connecteurs logiques	✓	✓		✓
Récursion structurelle	✓	✓	✓	✓
Récursion bien fondée	✓			
Compteur		✓		
Ordonnancement prémisses	✓	✓	✓	✓

✓ : fonctionne
$\sim$: fonctionne partiellement
[1] pour les fonctions complètes
[2] pour les fonctions non complètes
[3] mode $\{1\}$ ou $\{2\}$ mais pas $\{1, 2\}$

TABLE 9.1 – Fonctionnalités implantées

conséquents pour espérer pouvoir généraliser notre approche, nous avons tenu à tester le plugin d'extraction vers OCaml pour Coq sur des spécifications provenant d'autres projets :

— la sémantique à petit pas d'un langage monadique de MLCompCert [Dargaye 2009], un compilateur certifié pour un langage fonctionnel, dont nous extrayons un évaluateur,

— le comparatif "List Machine" de [Appel et al. 2012], dont nous extrayons l'interprète d'une étape d'exécution d'un langage assembleur,

— un interprète extrait d'une sémantique à petits pas de Featherweight Java, adaptée à partir d'une sémantique à grands pas [Kästner and Apel 2008].

Le code de MLCompCert que nous utilisions ne fonctionne plus avec la version 8.4 de Coq, qui est nécessaire pour faire fonctionner la dernière version de notre outil. Néanmoins, l'ancienne version de notre outil permettait d'extraire la sémantique d'un langage relativement conséquent. Cette dernière est archivée dans l'annexe B.

Parmi les exemples plus récents figure le code issu du comparatif "List Machine" qui contient la sémantique d'exécution d'un programme définie par les relations inductive `step` (une étape de l'exécution d'un programme C) et `run` (exécution d'un programme C) ainsi qu'une version exécutable : une fonction `exec`. La fonction est écrite sous forme d'un

`Fixpoint` et utilise un compteur qui décroît à chaque appel pour simuler la terminaison. Notre outil est capable d'extraire deux fonctions OCaml, qui permettent l'exécution des relations `step` et `run`.

Le tableau 9.2 montre les résultats obtenus pour ces trois exemples. Il indique pour chacun d'entre eux le nombre de relations inductives présentes dans la spécification, le temps nécessaire au processus d'extraction, le nombre de constructeurs de la plus grosse relation inductive présente dans la spécification ainsi que le temps d'extraction pour cette relation en particulier. Le tableau indique également les optimisations qui sont utilisées dans chaque exemple.

	MLCC	F. Java	List-machine
Nombre de relations inductives	4	4	3
Temps d'extraction	1 min	< 1 s	< 1 s
Nombre de constructeurs[1]	6	3	7
Temps d'extraction[1]	1 min	< 1 s	< 1 s
Fusion de constructeurs	✗	✓	✓
Connecteurs logiques	✓	✓	✗
Prédicat d'égalité	✓	✓	✓
Non déterminisme	✗	✓	✓

[1] De la plus grosse relation inductive dans la spécification.

TABLE 9.2 – Résultats pour trois extractions vers OCaml sur des outils provenant d'autres projets

9.2 Comparaison avec d'autres outils

Dans cette section, nous proposons un comparatif des performances concernant l'exécution des relations inductives des trois outils Coq, Isabelle/HOL et Twelf. Ces trois outils ont un fonctionnement très différent, et ne disposent pas des mêmes fonctionnalités en ce qui concerne les relations inductives. Le premier travail est de dégager des critères pertinents à évaluer. Ensuite nous réalisons une comparaison qualitative des trois outils dans la section 9.2.1, puis nous comparons les performances des outils en terme de vitesse d'extraction ou d'exécution du code obtenu dans la section 9.2.2.

Critères à évaluer Les outils que nous voulons comparer étant très différents, un test de performance seul ne suffit pas. Nous commençons par l'analyse d'une série de critères qualitatifs, qui mettent en avant les possibilités offertes par chacun des outils. Nous effectuons ensuite un test de performance, dont les résultats doivent être nuancés en fonction des critères qualitatifs. En effet, le test de performance est réalisé sur une spécification particulière, écrite de façon à pouvoir être traitée par les trois outils. Le nôtre étant le plus restrictif, il est aussi plus optimisé pour traiter ce type de spécifications.

9.2.1 Comparaison qualitative

Dans un premier temps, nous comparons les fonctionnalités d'extraction offertes par les trois outils dans le tableau 9.3. Par fonctionnalités d'extraction, nous entendons :

— Est-il possible d'extraire du code ?

— Le code extrait a t-il une forme lisible, est-il proche du code que l'on écrirait à la main ?

Twelf ne permet pas d'extraire du code, mais seulement d'exécuter des spécifications. En revanche Coq et Isabelle/HOL permettent l'extraction de code ML. Notre outil produit cependant du code lisible, alors qu'Isabelle/HOL produit du code à partir d'un encodage des inductifs, très éloigné du code qu'écrirait naturellement un utilisateur. La taille du code produit par Isabelle/HOL est également plus importante.

	Extraction	Code lisible
Coq	✓	✓
Isabelle/HOL	✓	
Twelf		

TABLE 9.3 – Fonctionnalités d'extraction

Un autre point important est la classe des spécifications traitées par les différents outils. Le tableau 9.4 indique que Twelf et Isabelle/HOL ont peu de restrictions au niveau des spécifications. Lorsque qu'une spécification n'est pas déterministe, l'exécution renvoie une liste de valeurs. Notre outil, en revanche, ne traite que les spécifications déterministes. Quand il n'est pas capable de reconnaître une spécification, il rejette simplement l'extraction. Pour cette raison, il ne traite qu'un sous-ensemble des relations inductives.

	Classe de relations traitées
Coq	limitée
Isabelle/HOL	non limitée
Twelf	non limitée

TABLE 9.4 – Classes de relations inductives traitées

Un point qui mérite d'être mentionné est la possibilité d'exécuter ou non le code extrait dans le formalisme initial. Nous distinguons deux niveaux dans le tableau 9.5 :

— La fonction extraite est-elle utilisable dans le même fichier source où se trouve la relation inductive ?

— La fonction extraite est-elle construite avec les mêmes constructions que la relation inductive ? En d'autres termes, est-il possible d'écrire des preuves liant la fonction extraite à la relation inductive ?

Le symbole $\sim$ pour Coq signifie qu'il est possible d'extraire du code vers Coq, au sein du même fichier, mais uniquement pour certaines spécifications car certaines fonctionnalités sont désactivées dans ce cas, comme cela est précisé dans le tableau 9.1.

	Exécution dans le f. source	Dans le formalisme initial
Coq	$\sim$	$\sim$
Isabelle/HOL	✓	
Twelf	✓	✓

TABLE 9.5 – Fonctionnalités d'exécution

Enfin, nous comparons dans le tableau 9.6 les fonctionnalités annexes de l'extraction. Twelf permet de vérifier si la spécification est complète [1] pour un mode d'exécution donné, et si cette exécution renvoie systématiquement ou non un résultat unique. Notre outil n'extrait du code qu'à partir de spécifications déterministes. Si l'extraction fonctionne, l'unicité est acquise. En ce qui concerne la complétude, elle est vérifiée au moment où la fonction est produite, car il est nécessaire de changer le type de retour de la fonction extraite. Le symbole $\sim$ indique que notre outil ne fournit pas de commandes spécifiques (contrairement à Twelf) pour effectuer ces tests en dehors du processus d'extraction. Isabelle/HOL ne permet pas d'effectuer ces tests.

1. La spécification est complète pour un mode si une sortie existe quelles que soient les entrées.

	Vérification de l'unicité	Vérification de la complétude
Coq	$\sim$	$\sim$
Isabelle/HOL		
Twelf	✓	✓

TABLE 9.6 – Fonctionnalités supplémentaires

9.2.2 Performances

Afin de comparer les performances d'extraction et d'exécution des différents outils, nous mesurons le temps d'extraction et d'exécution (du code extrait ou bien directement de l'exécution dans le cas de Twelf) d'une même spécification. Pour assurer une certaine objectivité, nous utilisons l'outil Ott [Sewell et al. 2010], qui permet, à partir d'une spécification écrite dans un langage spécifique, de générer du code utilisable pour plusieurs outils de preuve dont font partie Coq, Isabelle/HOL et Twelf.

9.2.2.1 Exemple utilisé

Nous utilisons la spécification de la sémantique du langage IMP, dont il est question dans l'introduction, dans sa totalité. La syntaxe du langage est donnée dans le tableau 9.7.

Le tableau 9.8 décrit la notion d'environnement nécessaire à l'exécution du langage IMP et introduit 5 relations inductives : $Jlookup$ et $Jchange$ qui permettent de manipuler l'environnement, $JevalA$ et $JevalB$ qui caractérisent l'évaluation des expressions arithmétiques et booléennes, et $Jexec$ qui caractérise l'exécution du langage IMP.

Nous donnons ici la définition de la relation d'exécution dans le tableau 9.9. La spécification complète, avec les 5 relations inductives, soit 46 règles au total, est donnée dans l'annexe C.

Pour mesurer le temps d'exécution de l'interprète, nous exécutons avec ce dernier un programme qui détermine si un nombre naturel est premier ou non. Ce programme est donné dans la figure 9.1.

n	::=		
	|	$\mathbf{O}$	
	|	$\mathbf{S}\ n$	
a	::=		
	|	n	
	|	x	
	|	$a + a'$	
	|	$a - a'$	
	|	$a * a'$	
	|	$a \% a'$	
	|	(a)	S
b	::=		
	|	$\mathbf{true}$	
	|	$\mathbf{false}$	
	|	$a = a'$	
	|	$a < a'$	
	|	$!b$	
	|	$b \wedge b'$	
	|	$b \vee b'$	
	|	(b)	S
c	::=		
	|	$\mathbf{skip}$	
	|	$x := a$	
	|	$c; c'$	
	|	$\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'$	
	|	$\mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}$	
	|	(c)	S

TABLE 9.7 – Syntaxe du langage IMP

s	::=		
	|	**empty**	
	|	$x\ n : s$	
Jlookup	::=		
	|	$x\ \mathbf{in}\ s \to n$	Lookup
Jchange	::=		
	|	$s[x/n] \to s'$	Change
Jeval	::=		
	|	$\langle a, s\rangle \rightsquigarrow n$	Evaluation
	|	$\langle b, s\rangle \rightsquigarrow b'$	Evaluation
Jexec	::=		
	|	$\langle c, s\rangle \longrightarrow s'$	Execution

TABLE 9.8 – Relations définissant l'exécution du langage IMP

```
if x0 < S S O then x2 := O
else if x0 = S S O then x2 := S O
else if x0 % S S O = O then x2 := O
else (
  ( x1 := S S S O;
    x2 := S O
  );
  while x1 * x1 < x0 + S O do
  (if x0 % x1 = O then x2 := O else skip);
  x1 := x1 + S S O od
)
```

FIGURE 9.1 – Programme de test IMP

$\boxed{\langle c, s\rangle \longrightarrow s'}$ Execution

$$\frac{}{\langle \mathbf{skip}, s\rangle \longrightarrow s} \quad \text{EX\_Skip}$$

$$\frac{\langle a, s\rangle \rightsquigarrow n \quad s[x/n] \to s'}{\langle x := a, s\rangle \longrightarrow s'} \quad \text{EX\_Affect}$$

$$\frac{\langle c, s\rangle \longrightarrow s' \quad \langle c', s'\rangle \longrightarrow s''}{\langle c; c', s\rangle \longrightarrow s''} \quad \text{EX\_Seq}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{true} \quad \langle c, s\rangle \longrightarrow s'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c', s\rangle \longrightarrow s'} \quad \text{EX\_Test1}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{false} \quad \langle c', s\rangle \longrightarrow s'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c', s\rangle \longrightarrow s'} \quad \text{EX\_Test2}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}, s\rangle \longrightarrow s} \quad \text{EX\_Loop1}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{true} \quad \langle c, s\rangle \longrightarrow s' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}, s'\rangle \longrightarrow s''}{\langle \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}, s\rangle \longrightarrow s''} \quad \text{EX\_Loop2}$$

Table 9.9 – Spécification de la relation *exec* du langage IMP écrite en Ott

9.2.2.2 Procédure de test

Initialement, le but de ces tests était de comparer la vitesse d'extraction d'Isabelle/HOL (version 2013-2) et de Coq (version 8.4pl3) et d'exécution du code extrait par ces deux outils avec la vitesse d'exécution de Twelf (version 1.7.1). Les trois outils que nous testons ont montré certaines limites. Pour Coq, nous n'avons pas rencontré de problème particulier. Pour Isabelle/HOL, l'extraction fonctionne, mais le programme extrait ne permet de faire que des calculs extrêmement simples. En revanche, nous avons pu exécuter le programme dans Isabelle/HOL à l'aide de la commande `values` qui permet de calculer des valeurs sans passer par un processus d'extraction. Nous avons de plus utilisé une optimisation précisant que nous n'attendons qu'une seule valeur (ce qui est raisonnable puisque la sémantique du langage IMP est déterministe). Twelf permet l'exécution de notre programme tant que la valeur de l'entier testé n'est pas trop grande.

Étant donné que nous testons des procédés différents pour chacun des outils, un programme extrait pour Coq et simplement l'exécution pour les deux autres, il faut trouver une grandeur à mesurer qui fasse du sens. Le but, lors d'un développement classique, étant généralement d'obtenir quelques valeurs, nous proposons de mesurer le temps pour obtenir ces valeurs. Pour Coq, nous mesurons le temps de compilation du fichier source Coq, l'extraction du programme OCaml, la compilation de ce dernier et l'exécution pour obtenir 5 valeurs. Dans le cas de Isabelle/HOL, nous chronométrons le temps de compilation du fichier source qui contient 5 commandes `values`. Enfin, nous mesurons le temps d'exécution de Twelf et demandons également le calcul de 5 valeurs.

La valeur que nous calculons est l'environnement qui résulte de l'exécution du programme de la figure 9.1. Afin de tester les limites des outils, nous faisons varier la valeur de la variable `x0` dans l'environnement initial, que nous nommons n. Le temps a été mesuré pour chaque valeur de n en faisant la moyenne de 10 tests complets (compilation + 5 exécutions).

9.2.2.3 Résultats

Tous les tests ont été effectués sur un ordinateur portable standard (processeur à deux cœurs cadencés à 2,5GHz et 8GB de mémoire vive). Le tableau 9.10 récapitule l'ensemble des résultats obtenus. L'absence de valeur, symbolisée par un tiret, signifie que l'outil a été incapable de terminer le calcul à la suite d'une erreur provoquée par une allocation de mémoire trop importante. Les deux courbes de la figure 9.2 montrent ces mêmes résultats avec deux échelles différentes pour les valeurs de n.

n	2	23	53	101	127	151	173
Coq	1,03	1,01	1,01	1,02	1,02	1,13	1,04
Isabelle/HOL	18,11	17,98	18,86	21,08	24,85	27,25	33,47
Twelf	0,03	0,31	1,95	5,31	11,7	13,07	20,06
n	199	227	251	277	307	331	353
Coq	1,12	1,1	1,13	1,06	1,07	1,12	1,08
Isabelle/HOL	37,81	48,57	50,85	55,23	80,37	82,75	84,29
Twelf	19,6133	-	-	-	-	-	-
n	373	401	421	449	479	499	557
Coq	1,13	1,17	1,42	1,21	1,16	1,13	1,21
Isabelle/HOL	116,5	118,64	133,74	196,67	195,93	197,35	-
Twelf	-	-	-	-	-	-	-

TABLE 9.10 – Vitesses d'exécution comparées

9.2.2.4 Interprétation des résultats

Pour $n = 0$, on observe que le plus rapide des outils est Twelf. Cela est dû au fait qu'il se lance très rapidement et qu'il n'y a aucun processus de compilation ou d'encodage des relations inductives. Pour Coq, le processus de compilation prend du temps car il fait intervenir les compilateurs Coq et OCaml avant de commencer l'exécution. Isabelle/HOL est le plus lent des trois car il procède à un encodage des relations inductives au moment de leur définition, puis il réalise de nouveaux calculs lors d'une commande qui les rend exécutables.

En ce qui concerne l'exécution avec des valeurs de n plus grandes, Isabelle/HOL et Twelf sont beaucoup plus longs que Coq. En effet, lorsque le nombre de règles (constructeurs des relations inductives) à appliquer pour obtenir une valeur augmente, les temps d'exécution

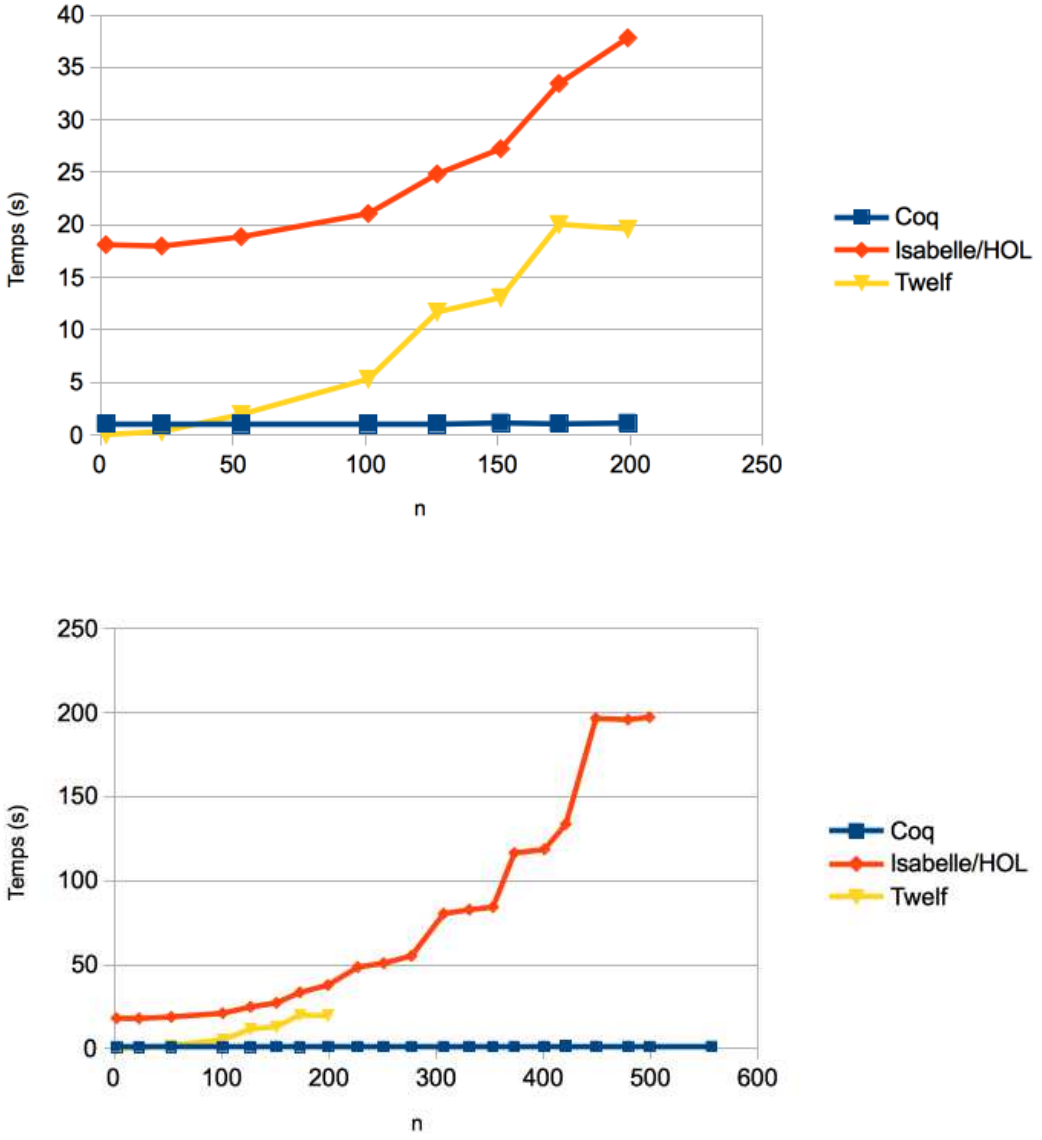


FIGURE 9.2 – Performance comparée de Coq, Isabelle/HOL et Twelf

deviennent problématiques. De plus, l'exécution dans ces outils demande une grande quantité de mémoire vive, ce qui les empêche de terminer les calculs pour des valeurs de n trop grandes. Twelf fait une erreur d'allocation de mémoire pour $n = 227$ et Isabelle/HOL pour $n = 557$. Le temps total qui est rapporté pour Coq varie très peu, car le temps d'exécution est négligeable devant le temps de compilation.

Nous avons montré, par ce comparatif, que notre outil apporte un moyen efficace pour tester et animer des spécifications complexes, à condition qu'elles soient déterministes. En effet, notre outil ne traite que des spécifications déterministes mais il a l'avantage de produire des fonctions qui s'exécutent aussi rapidement que des programmes fonctionnels classiques. Twelf montre de très bonnes performances pour travailler avec des exemples de taille raisonnable. En revanche, il faut 17,07 secondes à Isabelle/HOL pour compiler la sémantique du langage IMP et rendre la relation inductive exécutable, sans calculer aucune valeur. Enfin, notre outil semble être le seul à pouvoir produire du code éventuellement utilisable dans un environnement de production car il est le seul à offrir un temps d'exécution honorable avec des valeurs réalistes.

Conclusion

Bilan Notre approche consiste à produire du code fonctionnel à partir de relations inductives. Elle a donné lieu à l'élaboration d'un formalisme décrivant le processus de génération de code et trois implantations qui partagent le même code en ce qui concerne la génération de code fonctionnel.

Ces implantations offrent déjà des fonctionnalités intéressantes et permettent d'obtenir des résultats pour des spécifications assez réalistes, même s'il reste des extensions à développer pour qu'elles deviennent des outils complets. Les performances de notre mécanisme d'extraction et celles du code extrait sont encourageantes et sont bien supérieures à celles d'outils existants, comme l'extraction de code d'Isabelle/HOL qui utilise une approche très similaire à la nôtre.

Nous nous démarquons des autres approches, plus générales, par le fait de n'extraire que des fonctions renvoyant un seul résultat à partir de spécifications déterministes. Nous pensons que cette particularité est appropriée lors de l'utilisation d'outils d'aide à la preuve où le fait de vérifier systématiquement le déterminisme constitue, dans certains cas, une aide pour l'utilisateur. Dans le cas où l'on souhaite raisonner sur une spécification non déterministe, le fait d'extraire un programme fonctionnel fait moins de sens, dans la mesure où la fonction renvoie une liste de valeurs et n'est donc pas utilisable directement, à moins éventuellement de choisir systématiquement le premier résultat. De plus, le fait de se limiter à des relations déterministes permet des optimisations qui rendent le processus d'extraction ainsi que le code généré plus efficace.

Par rapport aux outils spécialisés, qui permettent de produire automatiquement une collection d'outils très élaborés à partir de la syntaxe et de la sémantique d'un langage, nous gardons la possibilité de raisonner sur le langage. En effet, l'étude des langages dans le cadre d'un outil d'aide à la preuve ne se limite pas à la génération d'un interprète, mais consiste à dégager et prouver des propriétés sur ces langages, comme par exemple la correction du typage par rapport à la sémantique. XText [Eclipse.org 2013] est l'un des outils les plus avancés en ce qui concerne la production automatique d'environnements pour les langages dédiés. Cependant, même s'il fait appel aux mêmes notions que notre outil, et s'il génère des interprètes, il s'adresse à un public différent. Il fournit un environnement pratique pour les utilisateurs d'un nouveau langage alors que nous nous adressons aux personnes voulant

formaliser un langage et prouver des propriétés sur ce langage ou sur les programmes qu'il permet d'exprimer.

Par rapport aux autres approches qui sont développées dans des outils d'aide à la preuve comme Isabelle/HOL ou Twelf, nous adoptons un point de vue pragmatique. Nous apportons une solution générale (exportable à d'autres outils) à un problème précis, et nous assurons des performances qui permettent une utilisation pratique même avec des spécifications conséquentes. Le problème de performance a déjà été soulevé pour Twelf dans [Appel et al. 2012]. Twelf permet néanmoins d'obtenir des résultats intéressants dans un certain nombre de cas. En revanche, Isabelle/HOL, même en dehors des considérations d'extraction, est assez lent ne serait-ce que pour définir des relations inductives. Il faut toutefois garder à l'esprit que ces outils essaient de résoudre un problème plus général. En cela, ils sont sans doute plus accessibles pour certains utilisateurs qui veulent exécuter des spécifications sans avoir à se soucier du déterminisme par exemple.

Pour les utilisateurs de Coq et de Focalize, nous apportons un outil nouveau qui facilite le développement de programmes certifiés en permettant aux utilisateurs de s'affranchir d'une quantité de travail considérable, mais aussi en facilitant l'évolution de ces programmes. En effet, nous pensons que la possibilité de générer automatiquement la version fonctionnelle d'une spécification et les preuves de correction offre au développeur la possibilité de faire évoluer plus rapidement la spécification, et de tester au plus tôt les effets de ces changements.

Travaux futurs Les travaux futurs concernent trois points qui sont :

— les extensions du processus d'extraction décrit dans le chapitre 5,

— la formalisation de ce processus d'extraction,

— l'extraction de relations inductives en général, sans les contraintes imposées dans ces travaux.

Commençons par détailler les optimisations qu'il reste à implanter dans le plugin Coq. Ces optimisations ne semblent pas présenter de difficultés majeures. Elles n'ont pas été implantées essentiellement faute de temps.

Concernant l'extraction vers OCaml, les fonctions en sortie de prémisses et les problèmes de non linéarité entre plusieurs prémisses peuvent se résoudre de la même façon. Pour une variable déjà définie dans une prémisse précédente (`x` déjà connue) on génère le code suivant :

```
match ... with
  | x' → if x = x' then ... else assert false
```

Et dans le cas d'une fonction `f` en sortie de prémisse on génère le code suivant :

```
| x → if x = f ... then ... else assert false
```

Le problème, c'est que si d'autres cas sont présents dans le filtrage cette astuce cesse rapidement de fonctionner. En effet, il faudra alors essayer les autres cas du filtrage, et ce après avoir sélectionné ce motif particulier. Il faut donc étudier les conditions sous lesquelles on peut appliquer cette optimisation.

Quant à l'extraction vers Coq, les extensions concernent le prédicat d'égalité, la terminaison et la génération des preuves.

Prédicat d'égalité Le prédicat d'égalité ne pose aucun problème dans le cas de l'extraction vers OCaml car nous supposons l'égalité décidable et nous traduisons l'égalité de Coq simplement vers l'égalité de OCaml. Cependant, dans le cas de la compilation vers le CIC de Coq, cela n'est plus possible. Il est nécessaire d'avoir une égalité décidable pour le type sur lequel s'applique l'égalité. Cette décidabilité doit être prouvée. Nous proposons deux solutions dans la section 7.5 : générer un lemme de décidabilité, ou bien utiliser un lemme de décidabilité écrit par l'utilisateur.

Terminaison Le recours au compteur permet de pouvoir extraire des spécifications Coq sans avoir à prouver la terminaison. Néanmoins, à terme, l'idéal serait de pouvoir extraire automatiquement des fonctions utilisant la récursion bien fondée, en demandant à l'utilisateur de fournir l'ordre. Se pose alors la question de la structure vers laquelle on souhaite compiler. En effet, des extensions de Coq telles que **`Function`** ou **`Program`** facilitent l'écriture des fonctions récursives, mais elles ne sont pas immédiatement compatibles avec notre technique de génération des preuves de correction. Nous pourrions en revanche nous inspi-

rer du travail réalisé par ces extensions et l'adapter pour prouver la terminaison en gardant le processus actuel de génération de code.

Preuves Pour l'instant, les preuves ne sont implantées que pour des cas relativement simples. La génération de preuves de correction ne supporte pas encore les cas suivants :

— fonctions partielles,
— fonctions extraites en mode complet,
— fonctions avec compteur,
— connecteurs logiques dans la spécification.

De plus, les preuves de complétude, quant à elles, ne sont pas encore implantées.

Pour les fonctions partielles, l'algorithme de génération de preuves est quasiment terminé. En réalité, nous savons déjà générer un script de preuve textuel, qui peut être copié dans le fichier Coq. Le problème vient du comportement d'une tactique OCaml qui n'est pas assez similaire à celui de la tactique en syntaxe concrète.

Dans certains de ces cas, des preuves manuelles ont déjà été réalisées et des schémas de preuve généraux assez similaires à celui du cas le plus basique semblent se dessiner.

Nous décrivons maintenant les pistes pour des développements ultérieurs plus conséquents, ou qui demandent une refonte du cadre que nous avons défini en formalisant notre approche dans le chapitre 5.

Extraction de code à partir de spécifications co-inductives Les types co-inductifs peuvent être utilisés pour spécifier des sémantiques de langages, permettant de caractériser les cas où l'interprétation du langage diverge [Leroy and Grall 2009]. De manière générale, les co-inductifs permettent d'exprimer des types dont les éléments peuvent être infinis, ou de définir des relations inductives qui ne terminent pas. Parallèlement, les **`CoFixpoint`** définissent des fonctions constructives, qui ne terminent pas forcément, mais dans lesquelles le résultat d'un appel récursif ne peut être utilisé que comme argument du résultat renvoyé par la fonction. Les co-inductifs sont suffisamment proches des inductifs pour que l'on envisage d'en extraire des fonctions vers OCaml et **`CoFixpoint`**. En revanche, les preuves semblent, quant à elles, bien plus compliquées à généraliser.

Déterminisme Voici quelques idées pour une meilleure détection du déterminisme. Il est possible de mettre en place un système plus souple pour vérifier le déterminisme lors de l'extraction à partir de relations inductives. Tout d'abord, la structure arborescente que constitue les Reltree est certes très pratique pour la génération de code mais elle est aussi plus restrictive que nécessaire. Par exemple, la fusion entre deux constructeurs ne permet de fusionner que des prémisses identiques. Pourtant, les prémisses qui effectuent un calcul pour définir une nouvelle variable n'ont pas besoin d'être identiques : elles peuvent très bien calculer deux valeurs différentes pour la même variable. La structure d'arbres pourrait être remplacée par des graphes orientés vérifiant certaines propriétés. Les chemins dans le graphe représenteraient alors les chemins d'exécution de la fonction, comme pour les Reltree.

Intégration dans d'autres outils de preuve Matita [Asperti et al. 2011] et Epigram [McBride 2004] sont deux outils qui peuvent constituer de bons candidats pour l'implantation de notre approche. Matita est un outil d'aide à la preuve similaire à Coq, qui est aussi basé sur le calcul des constructions inductives. Le gain que notre approche apporterait à Matita serait similaire à l'apport obtenu pour Coq. Epigram est un langage de programmation fonctionnel dans lequel il y a des types dépendants. Son objectif est de permettre une transition entre les langages de programmation classiques et les langages qui permettent d'intégrer des spécifications et des preuves, le système de types d'Epigram étant suffisant pour exprimer des spécifications de programmes. Ces spécifications sont suffisamment proches de celles de Focalize pour que l'on puisse utiliser notre approche au sein d'Epigram de la même manière que dans Focalize, c'est-à-dire en utilisant un ensemble de propriétés choisies par l'utilisateur comme la définition d'une relation inductive, même si cette dernière n'existe pas à proprement parler dans Epigram. Epigram permet aussi de définir des familles de types inductives, qui peuvent être vues comme des prédicats inductifs, il y a donc deux possibilités pour intégrer notre outil dans ce langage, suivant la manière dont on choisit de définir des relations inductives.

Bibliographie

Jean-Raymond Abrial. *The B Book, Assigning Programs to Meanings* Cambridge University Press, Cambridge (UK), 1996. 41

Andrew W. Appel, Robert Dockins, and Xavier Leroy. A List-Machine Benchmark for Mechanized Metatheory. *J. Autom. Reasoning*, 49(3) :453–491, 2012. 36, 138, 154

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita Interactive Theorem Prover. In *CADE*, pages 64–69, 2011. 157

Isabelle Attali and Didier Parigot. Integrating Natural Semantics and Attribute Grammars : the Minotaur System. Rapport de recherche RR-2339, INRIA, 1994. URL `http://hal.inria.fr/inria-00077110`. 38, 39

Gilles Barthe and Pierre Courtieu. Efficient Reasoning about Executable Specifications in Coq. In *TPHOLs*, pages 31–46, 2002. 45

Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and Reasoning About Recursive Functions : A Practical Tool for the Coq Proof Assistant. In *FLOPS*, pages 114–129, 2006. 45, 51, 111

Ulrich Berger, Kenji Miyamoto, Helmut Schwichtenberg, and Monika Seisenberger. Minlog - A Tool for Program Extraction Supporting Algebras and Coalgebras. In *CALCO*, pages 393–399, 2011. 37

Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In *TYPES*, pages 24–40, 2000. 34, 39

Les nombres à la fin de chaque référence correspondent aux numéros de page où la référence est présente.

Stefan Berghofer and Markus Wenzel. Inductive Datatypes in HOL - Lessons Learned in Formal-Logic Engineering. In *TPHOLs*, pages 19–36, 1999. 34

Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning Inductive into Equational Specifications. In *TPHOLs*, pages 131–146, 2009. 34

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004. 116

Lorenzo Bettini, Dietmar Stoll, Markus Völter, and Serano Colameo. Approaches and Tools for Implementing Type Systems in Xtext. In *SLE*, pages 392–412, 2012. 40

Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reasoning*, 43(3) :263–288, 2009. 21, 48, 112

Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 151–165, Yerevan (Armenia), October 2007. Springer. 35, 128, 130

Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and V. Pascual. CENTAUR : The System. In *Software Development Environments (SDE)*, pages 14–24, 1988. 37

Lukas Bulwahn. Code Generation from Inductive Predicates in Isabelle/HOL, 2009. Diploma Thesis. 35

Albert J. Camilleri. A Hybrid Approach to Verifying Liveness in a Symmetric Multi-Processor. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '97, pages 49–67, London, UK, UK, 1997. Springer-Verlag. 20

Brian Campbell. An Executable Semantics for CompCert C. In *CPP*, pages 60–75, 2012. 21, 48

Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model Checking : Algorithmic Verification and Debugging. *Commun. ACM*, 52(11) :74–84, 2009. 29

Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A Simple Applicative Language : Mini-ML. In *LISP and Functional Programming*, pages 13–27, 1986. 30

Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2-3) : 95–120, February 1988. 105

Thierry Coquand and Christine Paulin. Inductively Defined Types. In *Conference on Computer Logic*, pages 50–66, 1988. 26

Patrick Cousot and Radhia Cousot. Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977. 29

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A Software Analysis Perspective. In *SEFM*, pages 233–247, 2012. 44

Nils Anders Danielsson. Operational Semantics Using the Partiality Monad. In *ICFP*, pages 127–138, 2012. 48

Zaunah Dargaye. *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*. PhD thesis, Université Paris 7, 2009. 35, 91, 138

Zaynah Dargaye. `http://gallium.inria.fr/~dargaye/source/Mml.html`, April 2013. 173

Nicolaas G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer Berlin Heidelberg, 1970. URL `http://dx.doi.org/10.1007/BFb0060623`. 19

David Delahaye. A Tactic Language for the System Coq. In Michel Parigot and Andrei Voronkov, editors, *LPAR*, volume 1955, pages 85–95, Reunion Island (France), November 2000. Springer. 120

David Delahaye, Catherine Dubois, and Jean-Frédéric Étienne. Extracting Purely Functional Contents from Logical Inductive Types. In *TPHOLs*, pages 70–85, 2007. 35, 38, 57, 60

David Delahaye, Catherine Dubois, and Pierre-Nicolas Tollitte. Génération de code fonctionnel certifié à partir de spécifications inductives dans l'environnement Focalize. In *JFLA*, pages 55–81, La Ciotat (France), January 2010. Hermann. 23, 36, 134

The Coq development team. *The Coq Proof Assistant Reference Manual*. LogiCal Project, 2004. Version 8.0. 106, 117

Véronique Donzeau-Gouge, Gérard Huet, Bernard Lang, and Gilles Kahn. Programming environments based on structured editors : the Mentor experience. Rapport de recherche RR-0026, INRIA, 1980. URL `http://hal.inria.fr/inria-00076535`. 37

Catherine Dubois and Richard Gayraud. Compilation de la sémantique naturelle vers ML. In *JFLA*, pages 51–72, 1999. 38

Eclipse.org. *Xtext, version 2.4*. Fondation Eclipse, March 2013. `http://www.eclipse.org/Xtext/`. 29, 39, 153

Chucky Ellison and Grigore Roşu. An Executable Formal Semantics of C with Applications. In *POPL'12*, pages 533–544. ACM, 2012. 39

Frank Pfenning and Carsten Schuermann. *Twelf User's Guide, version 1.4*, 2002. `http://www.cs.cmu.edu/~twelf/guide-1-4/`. 36

Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *ECOOP*, pages 67–90, 1998. 38

Herman Geuvers. Proof assistants : History, ideas and future. *Sadhana*, 34(1) :3–25, February 2009. 20

Stéphane Glondu. *Vers une certification de l'extraction de Coq*. PhD thesis, Université Paris 7, 2012. 35

Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Christine Paulin Sandrine Blazy and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, Rennes, France, 2013. Springer. 20

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. 34

Florian Haftmann and Tobias Nipkow. Code Generation via Higher-Order Rewrite Systems. In *FLOPS*, pages 103–117, 2010. 34

Thomas C. Hales. Introduction to the Flyspeck Project. In *Mathematics, Algorithms, Proofs*, 2005a. 20

Thomas C. Hales. A Proof of the Kepler Conjecture. *Ann. Math.*, 162 :1065, 2005b. 20

Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1) :143–184, 1993. 36

William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. Reprint of 1969 article. 20

http ://focalize.inria.fr/. *Focalize, version 0.8.0*. INRIA, January 2013. 23, 137

Gilles Kahn. Natural Semantics. In *STACS*, pages 22–39, 1987. 30

Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench : Rules for Declarative Specification of Languages and IDEs. In *OOPSLA*, pages 444–463, 2010. 39

Gary A. Kildall. A Unified Approach to Global Program Optimization. In *POPL*, pages 194–206, 1973. 29

Paul Klint. A Meta-Environment for Generating Programming Environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2) :176–201, 1993. 38

Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2 (2) :127–145, 1968. 38

Christian Kästner and Sven Apel. Type-Checking Software Product Lines - A Formal Approach. *Automated Software Engineering, International Conference on*, 0 :258–267, 2008. 138

David Kågedal and Peter Fritzson. Generating a Modelica Compiler from Natural Semantics Specifications. In *In Proceedings of the Summer Computer Simulation Conference*, 1998. 38

David Lazar, Andrei Arusoaie, Traian-Florin Serbanuta, Chucky Ellison, Radu Mereuta, Dorel Lucanu, and Grigore Rosu. Executing Formal Semantics with the K Tool. In *FM*, pages 267–271, 2012. 39

Fabrice Le Fessant and Luc Maranget. Optimizing Pattern Matching. In *ICFP*, pages 26–37, 2001. 107

Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7) :107–115, 2009. 20, 29

Xavier Leroy and Hervé Grall. Coinductive Big-Step Operational Semantics. *Inf. Comput.*, 207(2) :284–304, 2009. 156

Pierre Letouzey. A New Extraction for Coq. In *TYPES*, pages 200–219, 2002. 57, 99

Pierre Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004. 35

Pierre Letouzey. Extraction in Coq : An Overview. In *CiE*, pages 359–369, 2008. 100

Michael Leuschel and Michael J. Butler. ProB : an automated analysis toolset for the B method. *STTT*, 10(2) :185–203, 2008. 41

Manuel Maarek and Virgile Prevosto. FocDoc : The Documentation System of Foc. In *Calculemus*. LIP6, September 2003. 133

Conor McBride. Epigram : Practical Programming with Dependent Types. In *Advanced Functional Programming*, pages 130–170, 2004. 157

Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-B models. In *SoICT*, pages 179–188, 2011. 41

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 20

Sam Owre, Natarajan Shankar, John M. Rushby, and Dave W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. 36

Christine Paulin-Mohring. Extracting F(omega)'s Programs from Proofs in the Calculus of Constructions. In *POPL*, pages 89–104, 1989a. 99

Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, January 1989b. 35

Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML Programs in the System Coq. *J. Symb. Comput.*, 15(5/6) :607–640, 1993. 99

Peter Frizson. *Developing Efficient Language Implementations from Structural and Natural Semantics*, March 2006.
`http://www.ida.liu.se/labs/pelab/rml/`. 38

Mikael Pettersson. RML - A New Language and Implementation for Natural Semantics. In *PLILP*, pages 117–131, 1994. 38

Mikael Pettersson. A Compiler for Natural Semantics. In *CC*, pages 177–191, 1996. 38, 67

Frank Pfenning and Christine Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. In *Mathematical Foundations of Programming Semantics*, pages 209–228, 1989. 26

Gordon D. Plotkin. A Structural Approach to Operational Semantics, 1981. 30

Dag Prawitz. *Natural Deduction : A Proof-Theoretical Study*. Dover Publications, 1965. 30

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott : Effective Tool Support for the Working Semanticist. *J. Funct. Program.*, 20(1) :71–122, 2010. 40, 142

Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In *TPHOLs*, pages 28–32, 2008. 34

Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. The implementation of mercury, an efficient purely declarative logic programming language. In *ILPS Workshop : Implementation Techniques for Logic Programming Languages*, 1994. 37

Matthieu Sozeau. Subset Coercions in Coq. In *TYPES*, pages 237–252, 2006. 45

Arnaud Spiwack. An Abstract Type for Constructing Tactics in Coq. In *Proof Search in Type Theory*, Edinburgh, Royaume-Uni, 2010. URL `http://hal.inria.fr/docs/00/50/25/00/PDF/tactics.pdf`. 116

Robert F. Stärk. Input/Output Dependencies of Normal Logic Programs. *J. Log. Comput.*, 4(3) :249–262, 1994. 39

Delphine Terrasse. Encoding Natural Semantics in Coq. In *AMAST*, pages 230–244, 1995. 40

The Coq Development Team. *Coq, version 8.4*. INRIA, August 2012. `http://coq.inria.fr/`. 20

Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Producing Certified Functional Code from Inductive Specifications. In *CPP*, pages 76–91, 2012. 23

Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Relationextraction. `http://coq.inria.fr/pylons/pylons/contribs/view/RelationExtraction/v8.4`, 2013a. 22, 137

Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Relationextraction. https://github.com/picnic/RelationExtraction/, 2013b. 137

Alberto Verdejo and Narciso Martí-Oliet. Two Case Studies of Semantics Execution in Maude : CCS and LOTOS. *Formal Methods in System Design*, 27(1-2) :113–172, 2005. 36

Alberto Verdejo and Narciso Martí-Oliet. Executable Structural Operational Semantics in Maude. *J. Log. Algebr. Program.*, 67(1-2) :226–293, 2006. 36

Makarius Wenzel and Stefan Berghofer. *The Isabelle System Manual*. TU München, February 2013. http://isabelle.in.tum.de/. 34

Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, May 1994. 105

Alfred N. Whitehead and Bertrand Russell. *Principia mathematica* . Cambridge Univ. Press, Camb. :, 1957. 19

Freek Wiedijk. http://www.cs.ru.nl/~freek/100/, March 2013. 20

Glynn Winskel. *The Formal Semantics of Programming Languages - an Introduction*. Foundation of computing series. MIT Press, 1993. 30

Faqing Yang, Jean-Pierre Jacquot, and Jeanine Souquières. The Case for Using Simulation to Validate Event-B Specifications. In *APSEC*, pages 85–90, 2012. 41

Annexes

Annexe A

Sémantique du langage MFL

$$\text{Fail}\frac{}{\Delta,\ \Delta_f \vdash fail \rhd fail} \qquad \text{Var}\frac{(x, v) \in \Delta}{\Delta,\ \Delta_f \vdash x \rhd v}$$

$$\text{Constr0}\frac{}{\Delta,\ \Delta_f \vdash C^0 \rhd C^0} \qquad \text{Tuple}\frac{\Delta,\ \Delta_f \vdash e_1 \rhd v_1 \quad \ldots \quad \Delta,\ \Delta_f \vdash e_n \rhd v_n}{\Delta,\ \Delta_f \vdash (e_1,\ \ldots,\ e_n) \rhd (v_1,\ \ldots,\ v_n)}$$

$$\text{App\_constr}\frac{\Delta,\ \Delta_f \vdash (e_1,\ \ldots,\ e_n) \rhd (v_1,\ \ldots,\ v_n)}{\Delta,\ \Delta_f \vdash C(e_1,\ \ldots,\ e_n) \rhd C(v_1,\ \ldots,\ v_n)}$$

$$\text{App\_fun}\frac{\begin{array}{c}\Delta,\ \Delta_f \vdash (e_1,\ \ldots,\ e_n) \rhd (v_1,\ \ldots,\ v_n) \\ (f(x1,\ \ldots,\ x_n),\ e) \in \Delta_f \qquad \Delta, (x_1,\ v_1), \ldots, (x_n,\ v_n),\ \Delta_f \vdash e \rhd v\end{array}}{\Delta,\ \Delta_f \vdash f(e_1,\ \ldots,\ e_n) \rhd v}$$

$$\text{Match\_ok}\frac{\begin{array}{c}\Delta,\ \Delta_f \vdash e \rhd v \qquad\qquad filter_\Delta(v,\ pat_i) = \Delta_i \\ \forall j,\ 1 \leq j \leq i,\ filter_\Delta(v,\ pat_j) = fail \qquad \Delta, \Delta_i,\ \Delta_f \vdash e_i \rhd v_i\end{array}}{\Delta,\ \Delta_f \vdash match\ e\ with \mid pat_1 \rightarrow e_1\ \ldots \mid pat_n \rightarrow e_n \rhd v_i}$$

$$\text{Match\_fail}\frac{\Delta,\ \Delta_f \vdash e \rhd v \qquad \forall j,\ 1 \leq j \leq n,\ filter_\Delta(v,\ pat_j) = fail}{\Delta,\ \Delta_f \vdash match\ e\ with \mid pat_1 \rightarrow e_1\ \ldots \mid pat_n \rightarrow e_n \rhd fail}$$

$$filter_\Delta(v,\ pat) = \begin{cases} \text{si } mgu_\Delta(v, pat) = \Delta_p \text{ alors } \Delta_p \\ \text{sinon } fail \end{cases}$$

$$\text{Define\_fun}\frac{\Delta'_f = \Delta_f, (f(x_1,\ \ldots,\ x_n), e)}{\Delta_f \vdash \mathsf{fun}\ f(x_1,\ \ldots,\ x_n)\ \rightarrow\ e \rhd \Delta'_f}$$

$$\text{Eval\_fun}\frac{(),\ \Delta_f \vdash f(e_1,\ \ldots,\ e_n) \rhd v}{\Delta_f \vdash f(e_1,\ \ldots,\ e_n) \rhd v}$$

Δ et Δ_f sont respectivement les environnements pour les variables et pour les fonctions. La fonction mgu calcule l'unificateur le plus général. Les fonctions sont ajoutées dans Δ_f chaque fois que le mot-clé fun est rencontré, en utilisant la règle Define_fun. Enfin, la règle Eval_fun permet d'évaluer l'application d'une fonction.

Annexe B

Sémantique du sixième langage intermédiaire de MLCompCert

Cet exemple provient de la documentation du code source de MLCompCert [Dargaye 2013].

```
Section Fsem.

Variable s: list (ident*funct).

Inductive feval_atom:env→atom→fval→Prop:=
|fvar: forall e id v , var_test_prop id →
    PTree.get id e =Some v → feval_atom e (Fvar id) v
|ffield : forall e n fid t vl v,
    feval_atom e t (fclos fid vl) →
    nth_error vl n=Some v→
    feval_atom e (Ffield (S n) t) v.

Inductive feval_atlist: env→list atom→list fval→Prop:=
|fanil: forall e, feval_atlist e nil nil
|facons: forall e a v al vl, feval_atom e a v→feval_atlist e al vl→
 feval_atlist e (a::al) (v::vl).

Inductive feval_term:env→Fterm→fval→Prop:=
|fAtom : forall e a v,
       feval_atom e a v →
       feval_term e (Atom a) v
| fClos: forall e fid tl vl d,
       recup_funct fid s = Some d →
       feval_atlist e tl vl →
       feval_term e (Fclos fid tl) (fclos fid vl)
|fcon: forall e n tl vl,
       feval_atlist e tl vl →
       feval_term e (Fcon n tl) (fconstr n vl)
|flet: forall e id t1 v1 t2 v2,
```

```
       feval_term e t1 v1→ var_test_prop id →
       feval_term (PTree.set id v1 e) t2 v2 →
       feval_term e (Flet id t1 t2) v2
|fapp:forall e t1 fid vl targs vargs d e1 v info,
       feval_atom e t1 (fclos fid vl) →
       recup_funct fid s = Some d→ varl_test_prop (fun_par d) →
       list_norepet (fun_par d) →
       feval_atlist e targs vargs→
       set_local_par (fun_par d) ( (fclos fid vl)::vargs)
           (@PTree.empty fval) =Some e1→
       feval_term e1 (fun_body d) v →
       (info = None \/ info = Some fid) →
       feval_term e (Fapp info t1 targs) v
|fmatch : forall e a pl n vl xl p e1 v,
       feval_atom e a (fconstr n vl) →
       nth_error pl n = Some (FPatc xl p) →
       varl_test_prop xl →
       list_norepet xl →
       set_local_par xl vl e = Some e1→
       feval_term e1 p v →
       feval_term e (Fmatch a pl) v.

Inductive feval_list:env→list Fterm→list fval→Prop:=
|fnil: forall e, feval_list e nil nil
|fcons: forall e hd vhd tl vtl,
    feval_term e hd vhd →feval_list e tl vtl→
    feval_list e (hd::tl) (vhd::vtl).

End Fsem.
```

Annexe C

Spécification Ott du langage IMP

$$
\begin{array}{lll}
n & ::= & \\
 & | & \mathbf{O} \\
 & | & \mathbf{S}\ n \\
 & & \\
a & ::= & \\
 & | & n \\
 & | & x \\
 & | & a + a' \\
 & | & a - a' \\
 & | & a * a' \\
 & | & a \% a' \\
 & | & (a) \qquad \mathsf{S} \\
 & & \\
b & ::= & \\
 & | & \mathbf{true} \\
 & | & \mathbf{false} \\
 & | & a = a' \\
 & | & a < a' \\
 & | & !b
\end{array}
$$

	$b \wedge b'$	
	$b \vee b'$	
	(b)	S

c	::=	
	$\mid$	**skip**
	$\mid$	$x := a$
	$\mid$	$c; c'$
	$\mid$	**if** b **then** c **else** c'
	$\mid$	**while** b **do** c **od**
	$\mid$	(c) S
s	::=	
	$\mid$	**empty**
	$\mid$	$x\ n : s$

Jlookup	::=		
	$\mid$	x **in** $s \to n$	Lookup

Jchange	::=		
	$\mid$	$s[x/n] \to s'$	Change

Jeval	::=		
	$\mid$	$\langle a, s\rangle \rightsquigarrow n$	Evaluation
	$\mid$	$\langle b, s\rangle \rightsquigarrow b'$	Evaluation

Jexec	::=		
	$\mid$	$\langle c, s\rangle \longrightarrow s'$	Execution

$\boxed{x \textbf{ in } s \to n}$ Lookup

$$\frac{x = x'}{x \textbf{ in } x'\ n : s \to n}\ \text{L\_Lookup1}$$

$$\frac{x \neq x' \qquad x \textbf{ in } s \to n}{x \textbf{ in } x'\ n' : s \to n}\ \text{L\_Lookup2}$$

$\boxed{s[x/n] \to s'}$ Change

$$\frac{}{\textbf{empty}[x/n] \to x\ n : \textbf{empty}}\ \text{L\_Change1}$$

$$\frac{x = x'}{x'\ n' : s[x/n] \to x\ n : s}\ \text{L\_Change2}$$

$$\frac{x \neq x' \qquad s[x/n] \to s'}{x'\ n' : s[x/n] \to x'\ n' : s'}\ \text{L\_Change3}$$

$\boxed{\langle a, s\rangle \rightsquigarrow n}$ Evaluation

$$\frac{}{\langle n, s\rangle \rightsquigarrow n} \quad \text{EA\_Num}$$

$$\frac{x \,\mathbf{in}\, s \to n}{\langle x, s\rangle \rightsquigarrow n} \quad \text{EA\_Var}$$

$$\frac{\langle a, s\rangle \rightsquigarrow n \quad \langle a', s\rangle \rightsquigarrow \mathbf{O}}{\langle a + a', s\rangle \rightsquigarrow n} \quad \text{EA\_Plus1}$$

$$\frac{\langle a, s\rangle \rightsquigarrow n \quad \langle a', s\rangle \rightsquigarrow \mathbf{S}\, n' \quad \langle \mathbf{S}\, n + n', s\rangle \rightsquigarrow n''}{\langle a + a', s\rangle \rightsquigarrow n''} \quad \text{EA\_Plus2}$$

$$\frac{\langle a, s\rangle \rightsquigarrow n \quad \langle a', s\rangle \rightsquigarrow \mathbf{O}}{\langle a - a', s\rangle \rightsquigarrow n} \quad \text{EA\_Minus1}$$

$$\frac{\langle a, s\rangle \rightsquigarrow \mathbf{S}\, n \quad \langle a', s\rangle \rightsquigarrow \mathbf{S}\, n' \quad \langle n - n', s\rangle \rightsquigarrow n''}{\langle a - a', s\rangle \rightsquigarrow n''} \quad \text{EA\_Minus2}$$

$$\frac{\langle a', s\rangle \rightsquigarrow \mathbf{O}}{\langle a * a', s\rangle \rightsquigarrow \mathbf{O}} \quad \text{EA\_Mult1}$$

$$\frac{\langle a, s\rangle \rightsquigarrow n \quad \langle a', s\rangle \rightsquigarrow \mathbf{S}\, n' \quad \langle n * n', s\rangle \rightsquigarrow n'' \quad \langle n + n'', s\rangle \rightsquigarrow n'''}{\langle a * a', s\rangle \rightsquigarrow n'''} \quad \text{EA\_Mult2}$$

$$\frac{\langle a < a', s\rangle \rightsquigarrow \mathbf{true} \quad \langle a, s\rangle \rightsquigarrow n}{\langle a \% a', s\rangle \rightsquigarrow n} \quad \text{EA\_Modulo1}$$

$$\frac{\langle a < a', s\rangle \rightsquigarrow \mathbf{false} \quad \langle a - a', s\rangle \rightsquigarrow n \quad \langle n \% a', s\rangle \rightsquigarrow n'}{\langle a \% a', s\rangle \rightsquigarrow n'} \quad \text{EA\_Modulo2}$$

$\boxed{\langle b, s\rangle \rightsquigarrow b'}$ Evaluation

$$\frac{}{\langle \mathbf{true}, s\rangle \rightsquigarrow \mathbf{true}} \quad \text{EB\_True}$$

$$\frac{}{\langle \mathbf{false}, s\rangle \rightsquigarrow \mathbf{false}} \quad \text{EB\_False}$$

$$\frac{\langle a, s\rangle \rightsquigarrow \mathbf{O} \quad \langle a', s\rangle \rightsquigarrow \mathbf{O}}{\langle a = a', s\rangle \rightsquigarrow \mathbf{true}} \quad \text{EB\_Eq1}$$

$$\frac{\langle a, s\rangle \rightsquigarrow \mathbf{S}\ n \quad \langle a', s\rangle \rightsquigarrow \mathbf{O}}{\langle a = a', s\rangle \rightsquigarrow \mathbf{false}} \quad \text{EB\_Eq2}$$

$$\frac{\langle a, s\rangle \rightsquigarrow \mathbf{O} \quad \langle a', s\rangle \rightsquigarrow \mathbf{S}\ n}{\langle a = a', s\rangle \rightsquigarrow \mathbf{false}} \quad \text{EB\_Eq3}$$

$$\frac{\langle a, s\rangle \rightsquigarrow \mathbf{S}\ n \quad \langle a', s\rangle \rightsquigarrow \mathbf{S}\ n' \quad \langle n = n', s\rangle \rightsquigarrow b}{\langle a = a', s\rangle \rightsquigarrow b} \quad \text{EB\_Eq4}$$

$$\frac{\langle a, s\rangle \rightsquigarrow \mathbf{O} \quad \langle a', s\rangle \rightsquigarrow \mathbf{S}\ n}{\langle a < a', s\rangle \rightsquigarrow \mathbf{true}} \quad \text{EB\_Inf1}$$

$$\frac{\langle a, s\rangle \rightsquigarrow n \quad \langle a', s\rangle \rightsquigarrow \mathbf{O}}{\langle a < a', s\rangle \rightsquigarrow \mathbf{false}} \quad \text{EB\_Inf2}$$

$$\frac{\langle a, s\rangle \rightsquigarrow \mathbf{S}\ n \quad \langle a', s\rangle \rightsquigarrow \mathbf{S}\ n' \quad \langle n < n', s\rangle \rightsquigarrow b}{\langle a < a', s\rangle \rightsquigarrow b} \quad \text{EB\_Inf3}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{true}}{\langle !b, s\rangle \rightsquigarrow \mathbf{false}} \quad \text{EB\_Not1}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{false}}{\langle !b, s\rangle \rightsquigarrow \mathbf{true}} \quad \text{EB\_Not2}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{true} \quad \langle b', s\rangle \rightsquigarrow \mathbf{true}}{\langle b \wedge b', s\rangle \rightsquigarrow \mathbf{true}} \quad \text{EB\_And1}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{false} \quad \langle b', s\rangle \rightsquigarrow \mathbf{true}}{\langle b \wedge b', s\rangle \rightsquigarrow \mathbf{false}} \quad \text{EB\_And2}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{true} \quad \langle b', s\rangle \rightsquigarrow \mathbf{false}}{\langle b \wedge b', s\rangle \rightsquigarrow \mathbf{false}} \quad \text{EB\_And3}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{false} \quad \langle b', s\rangle \rightsquigarrow \mathbf{false}}{\langle b \wedge b', s\rangle \rightsquigarrow \mathbf{false}} \quad \text{EB\_And4}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{true} \quad \langle b', s\rangle \rightsquigarrow \mathbf{true}}{\langle b \vee b', s\rangle \rightsquigarrow \mathbf{true}} \quad \text{EB\_Or1}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{false} \quad \langle b', s\rangle \rightsquigarrow \mathbf{true}}{\langle b \vee b', s\rangle \rightsquigarrow \mathbf{true}} \quad \text{EB\_Or2}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{true} \quad \langle b', s\rangle \rightsquigarrow \mathbf{false}}{\langle b \vee b', s\rangle \rightsquigarrow \mathbf{true}} \quad \text{EB\_Or3}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{false} \quad \langle b', s\rangle \rightsquigarrow \mathbf{false}}{\langle b \vee b', s\rangle \rightsquigarrow \mathbf{false}} \quad \text{EB\_Or4}$$

$\boxed{\langle c, s\rangle \longrightarrow s'}$ Execution

$$\frac{}{\langle \mathbf{skip}, s\rangle \longrightarrow s} \quad \text{EX\_Skip}$$

$$\frac{\langle a, s\rangle \rightsquigarrow n \quad s[x/n] \to s'}{\langle x := a, s\rangle \longrightarrow s'} \quad \text{EX\_Affect}$$

$$\frac{\langle c, s\rangle \longrightarrow s' \quad \langle c', s'\rangle \longrightarrow s''}{\langle c; c', s\rangle \longrightarrow s''} \quad \text{EX\_Seq}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{true} \quad \langle c, s\rangle \longrightarrow s'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c', s\rangle \longrightarrow s'} \quad \text{EX\_Test1}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{false} \quad \langle c', s\rangle \longrightarrow s'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c', s\rangle \longrightarrow s'} \quad \text{EX\_Test2}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}, s\rangle \longrightarrow s} \quad \text{EX\_Loop1}$$

$$\frac{\langle b, s\rangle \rightsquigarrow \mathbf{true} \quad \langle c, s\rangle \longrightarrow s' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}, s'\rangle \longrightarrow s''}{\langle \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}, s\rangle \longrightarrow s''} \quad \text{EX\_Loop2}$$

le cnam

Pierre-Nicolas TOLLITTE

Extraction de code fonctionnel certifié à partir de spécifications inductives

le cnam

Résumé :
Les outils d'aide à la preuve basés sur la théorie des types permettent à l'utilisateur d'adopter soit un style fonctionnel, soit un style relationnel (c'est-à-dire en utilisant des types inductifs). Chacun des deux styles a des avantages et des inconvénients. Le style relationnel peut être préféré parce qu'il permet à l'utilisateur de décrire seulement ce qui est vrai, de s'abstraire temporairement de la question de la terminaison, et de s'en tenir à une description utilisant des règles. Cependant, une spécification relationnelle n'est pas exécutable.
Nous proposons un cadre général pour transformer une spécification inductive en une spécification fonctionnelle, en extrayant à partir de la première une fonction et en produisant éventuellement la preuve de correction de la fonction extraite par rapport à sa spécification inductive. De plus, à partir de modes définis par l'utilisateur, qui permettent de considérer les arguments de la relation comme des entrées ou des sorties (de fonction), nous pouvons extraire plusieurs comportements calculatoires à partir d'un seul type inductif.
Nous fournissons également deux implantations de notre approche, l'une dans l'outil d'aide à la preuve Coq et l'autre dans l'environnement Focalize. Les deux sont actuellement distribuées avec leurs outils respectifs.

Mots clés :
Spécifications exécutables, relations inductives, génération de code fonctionnel, génération de preuves, Coq, Focalize.

Abstract :
Proof assistants based on type theory allow the user to adopt either a functional style, or a relational style (e.g., by using inductive types). Both styles have advantages and drawbacks. Relational style may be preferred because it allows the user to describe only what is true, discard momentarily the termination question, and stick to a rule-based description. However, a relational specification is usually not executable.
We propose a general framework to turn an inductive specification into a functional one, by extracting a function from the former and eventually produce the proof of soundness of the extracted function w.r.t. its inductive specification. In addition, using user-defined modes which label inputs and outputs, we are able to extract several computational contents from a single inductive type.
We also provide two implementations of our approach, one in the Coq proof assistant and the other in the Focalize environnement. Both are currently distributed with the respective tools.

Keywords :
Executable specifications, inductive relations, functional code generation, proof generation, Coq, Focalize.