De nouveaux outils pour calculer avec des inductifs en Coq

Pierre Boutillier

▶ To cite this version:

Pierre Boutillier. De nouveaux outils pour calculer avec des inductifs en Coq. Langage de programmation [cs.PL]. Université Paris-Diderot - Paris VII, 2014. Français. NNT : . tel-01054723

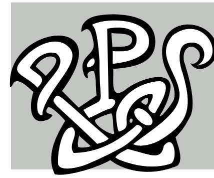
HAL Id: tel-01054723
https://theses.hal.science/tel-01054723

Submitted on 8 Aug 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.

Université Paris Diderot (Sorbonne Paris Cité)
Laboratoire PPS – Équipe πr^2
École Doctorale Sciences Mathématique de Paris Centre



Thèse de doctorat
spécialité Informatique

De nouveaux outils pour Calculer avec des inductifs en Coq

présentée et soutenue publiquement par

Pierre Boutillier

le 18 février 2014

devant le jury composé de

M.	Yves	Bertot	*rapporteur*
M.	Roberto	Di Cosmo	*président*
M.	Hugo	Herbelin	*directeur*
M.	Daniel	Hirschkoff	*examinateur*
M.	Conor	McBride	*examinateur*
Mme.	Christine	Paulin-Mohring	*examinatrice*
M.	Carsten	Schuermann	*rapporteur*

À Madou et tonton Jean-Pierre,

Mercis

Le grade de docteur ne s'obtient jamais seul. J'ai le privilège d'avoir été merveilleusement entouré.

Antichronologiquement, comme le veut la coutume, Yves Bertot et Carsten Schuermann ont dû endurer mon style et m'ont malgré tout autorisé à soutenir. Je leur suis énormément reconnaissant de l'effort d'une part et de la pertinence de leurs remarques d'autre part. D'autant que Carsten l'a fait dans une langue étrangère et Yves avec bienveillance.

Les discussions avec Hugo m'ont toujours énormément éclairé sur comment aborder une question en conservant ma manière de penser ; vision, il me semble, que nous partageons.

Ce manuscrit n'est « pas si pire » grâce à Pierre Letouzey qui m'a offert un peu de la rigueur qui fait tenir `Coq` debout pour relire exhaustivement ma thèse. Le reste de l'équipe πr^2 Yann, Matthieu, Alexis, Pierre-Louis, Pierre-Marie, Lourdes, Guillaume, Matthias, Stéphane, Vincent ont tous joué un rôle dans l'écriture et/ou la conception de cette thèse et de sa soutenance. Christine, Frédérique et Thomas ont aussi beaucoup aidé à la genèse de ce document.

Fernand Deligny (pédagogue) disait « Soit surtout présent lorsque tu n'es pas là. » C'est le tour de force que réussit Christine Paulin-Mohring avec l'équipe de développement de `Coq` ! Bien que nos échanges directs soient récents, son influence ne l'est pas et je la remercie chaleureusement d'évaluer mon travail.

Mon cerveau a été scientifiquement façonné par Conor McBride. Mes quelques idées originales trouvent leur source dans sa folie. Avant lui, Daniel Hirschkoff (avec l'ensemble de l'équipe PLUME) m'avait définitivement fait tomber dans les bras de la programmation comme support du raisonnement. Daniel fut aussi ma boussole aux moments clés de mon cheminement scientifique.

Daniel et Roberto Di Cosmo, qui m'honore de la présidence de mon jury, furent, de mes enseignants d'informatique, ceux qui m'ont le plus marqué. J'estime que l'enseignement fut une part importante de mon travail de thèse et je pense que si elle fut si plaisante, c'est grâce à leur modèle auxquel j'ai pu me raccrocher.

Mes études supérieures doivent énormément au cadre dans lequel elles ont eu lieu. Il me faut donc remercier l'ENS Lyon et le Lycée Fenelon (et donc ceux qui les font ainsi) pour m'avoir d'abord séduit au premier regard, puis pour m'être fait me dire que même rentré par la fenêtre, l'année d'après j'y serai et enfin pour ne m'avoir, mais alors pas du tout, déçu ensuite !

A tous les niveaux, les établissements que j'ai fréquentés tenaient debout grâce à leur responsable administratif. Merci pour tout ce travail de l'ombre !

Ma famille m'a donné la possibilité d'un épanouissement personnel rare. Mes parents ont réussi le tour de force de toujours m'encourager à être moi-même sans jamais abandonner l'idée d'améliorer mes torts. Je me suis construit grâce à leur persévérance. Toute la famille

m'a énormément soutenu et me dire « finis ta thèse pour qu'ils puissent un peu se dire que tu n'as pas gâché toute la liberté qu'ils t'ont ménagée » a été l'un des moteurs irrationnels forts pour mener l'aventure au bout. Sneusneur est aussi à un tournant qu'elle devrait surmonter très fort, nous avons toujours fonctionné en symbiose.

Ma thèse et beaucoup plus doivent quotidiennement énormément à Maud.

Table des matières

Introduction 9
Logiques constructives 9
Quantification 10
La généralisation : les inductifs 11
Correspondance preuves/programmes 11
Machines universelles 12
Programmation fonctionnelle avec types algébriques 12
Types riches 13
Utilisation des données inductives 14
Réduction des points fixes 15

1 Un langage fonctionnel avec types riches 17
1.1 Conventions d'écriture 17
1.2 Le cœur fonctionnel 18
1.2.1 Syntaxe 18
1.2.2 Evaluation 18
1.2.3 Typage 19
1.3 Déclarations globales 19
1.3.1 Syntaxe 19
1.3.2 Evaluation 20
1.3.3 Typage 20
1.4 Types de données algébriques strictement positifs 21
1.4.1 Syntaxe 21
1.4.2 Evaluation 23
1.4.3 Typage 24
1.5 Des structures de données d'exemple 24
1.6 Digression sur la représentation des constructeurs et des branches 26
1.7 Expressivité de l'analyse de cas 27

2 Le filtrage en programmation fonctionnelle 29
2.1 Principe du filtrage 30
2.2 Reconnaitre un motif 32
2.3 Construire un arbre de décision 33
2.3.1 Arbre de décision 33
2.3.2 Mécanique d'atomisation 34

3 Encodage du filtrage dans l'analyse de cas 37
3.1 Le filtrage dépendant primitif 37
3.2 Encodage du filtrage par des analyses de cas 38
3.2.1 Des types riches exprimant les problèmes d'unification 39
3.2.2 Manipulation des égalités d'index 40
3.3 Utilisabilité de l'encodage 41
3.4 Constructions génériques autour d'une analyse de cas dépendante 42
3.4.1 Coupures entrelacées 42
3.4.2 Elimination des branches impossibles 43

4 Du filtrage à l'analyse de cas structurellement 45
4.1 Squelette d'index 45
4.2 Quand les égalités sont inutiles 45
4.3 Diagonalisation 47
4.4 Quand le diagonaliseur en dit trop 52
4.5 Des clauses de retour pour éliminer tous les termes 53
4.5.1 Les index du type ne contiennent que des variables et des constructeurs 53
4.5.2 Hors du cadre pseudo-motif 54
4.6 Digression sur les cas impossibles dans des points fixes 55
4.7 Mécaniser la diagonalisation et compiler le filtrage 56
4.7.1 Construire une analyse de cas à partir de sa clause de retour 56
4.7.2 D'un arbre de décision à des analyses de cas 57
4.7.3 Extraire un diagonaliseur 59
4.8 Eléments de correction 60

5 Simplifier un terme 63
5.1 Normalisation en appel par nom 63
5.2 Trace des constantes dépliées 66
5.3 Refolding Algebraic Krivine Abstract Machine 69
5.4 Configurabilité 70
5.5 Discussion 73

6 Etablir qu'un point fixe ne produit pas de calcul infini 75
6.1 Décroissance structurelle 75
6.2 Raffinement lié à l'induction 79
6.3 Raffinement lié aux coupures entrelacées 81
6.4 Raffinement lié à la réduction 82
6.4.1 Garder la forme $\beta\iota\delta$ normale 82
6.4.2 Retrouver la réduction forte 84
6.4.3 Questions d'efficacité 84
6.5 La règle ϕ de réduction impose une garde structurelle 86

Conclusion et perspectives 89

Introduction

Aucune tâche n'est atomique. Toute action se décompose en opérations plus élémentaires. Agir demande en permanence de concevoir des *algorithmes* afin de réaliser des activités complexes. Le concept de *programme* et son *exécution* ou *calcul* est bien antérieur à l'apparition des ordinateurs. L'informatique, la science qui les étudie, n'a donc pas attendu des machines aussi complexes pour susciter l'intérêt. La mécanisation n'a fait que démultiplier le besoin d'y réfléchir formellement.

L'humain a le souci de l'optimisation. Dès qu'il trouve un algorithme, il se demande comment faire la même chose en moins d'étapes ou par d'autres moyens.

Pourtant, après modifications, l'algorithme réalise-t-il effectivement toujours « la même chose » ?

Nos cousins shadocks affirment que « Plus un ordinateur va vite, plus il donne de bons résultats… ». Laissons-leur l'ambiguïté sur la notion de bon résultat et le pied de nez que j'y vois à mon ministère de tutelle. Cette introduction établit un cadre mathématique qui aborde formellement la question de la *correction* (le fait d'être correct) d'un résultat. Ce manuscrit n'y répond pas ; il aborde des problématiques pour faciliter l'utilisation de systèmes qui permettent d'y répondre.

Nous allons d'abord énumérer des systèmes logiques dans lesquels prouver revient de plus en plus à manipuler des données et des fonctions. Il est légitime de parler de logique pour aborder la programmation et d'utiliser des structures de données pour appuyer un raisonnement logique. Nous basculerons en effet dans l'informatique en décrivant les découvertes qui ont identifié les deux problématiques en donnant, suivant le point de vue, aux preuves un corps ou aux programmes une description de leur comportement. Ainsi, nous aboutirons à un langage de programmation pratique qui apporte des garanties fortes sur la bonne formation des programmes qu'il permet d'écrire. Nous aurons alors le socle nécessaire pour discuter nos problématiques.

Logiques constructives

Formaliser la logique est apparu comme un champ des mathématiques au cours du 19ᵉ siècle. Des primitives autour desquelles s'articule le raisonnement ont émergé. Parmi ces *connecteurs* :

l'implication exprime la possibilité d'obtenir une propriété B à partir d'une propriété A et s'écrit $A \to B$. Elle vient avec l'axiome que pour toute propriété A, $A \to A$.

la conjonction fait de deux propriétés une propriété.

la disjonction introduit la notion de choix. Elle se construit en donnant l'une des deux possibilités. Elle demande de savoir agir dans les deux cas pour être détruite.

faux n'a pas de règle d'obtention. Elle permet de nier toute proposition.

Par ailleurs, les premiers systèmes formels venaient avec la supposition toujours faite jusque là que l'on peut toujours considérer qu'une propriété est soit vraie soit fausse (principe du tiers exclu). On parle aujourd'hui de logique *classique*.

Dans les logiques classiques, prouver l'impossibilité qu'il n'existe pas revient à prouver qu'il existe. L'objet dont la non existence a été niée n'est pas pour autant construit. Un mouvement s'est constitué pour établir des logiques dans lesquelles prouver qu'il existe impose d'exhiber un témoin d'existence. Ce schéma de pensée appelé *constructivisme* a un intérêt particulier pour voir les preuves d'existence comme des algorithmes de construction.

Quelles que soient les structures de données utilisées et les règles de raisonnement autorisées, il existe plusieurs manières de structurer un raisonnement. Parmi elles, la *déduction naturelle* définit les connecteurs par deux types de règles : des règles pour les construire ou introduire et des règles pour les détruire ou utiliser. Cette manière de faire est assez intuitive lors de la construction de preuves par l'esprit humain.

Le *calcul des séquents* est lui efficace lors du traitement mécanique des preuves. Pour chaque connecteur, il s'articule autour de règles dites gauches si on suppose avoir la donnée et de règles dites droites si on veut l'obtenir.

Quantification

Le type des entiers naturels, défini par l'*arithmétique de Peano*, représente une classe d'objet contenant une infinité d'habitants distincts directement manipulables dans la logique.

Les nombres sont construits à l'aide de deux primitives : une constante nommée *zéro* et une fonction injective *successeur* qui à tout entier associe l'entier suivant. Surtout, un nouveau concept arrive pour utiliser les nombres entiers : le *raisonnement par récurrence*. Il permet de prouver des propriétés qui dépendent d'un entier par un raisonnement potentiellement infini. Ce raisonnement est en réalité fini pour tout entier fini car il donne un comportement à adopter fini pour l'entier zéro et un comportement fini pour passer de la propriété pour un entier à cette propriété pour son successeur.

L'arithmétique permet d'énoncer des propriétés qui parlent d'entiers naturels. Elle dispose en plus de l'implication d'une notion de *quantification* : une propriété peut parler des valeurs des objets qu'elle suppose. On peut écrire, pour tout entier naturel n, la propriété P dans laquelle n apparaît.

La quantification est aussi utilisable pour permettre à une propriété de dépendre d'une autre propriété. Par ce biais, une propriété peut parler de toutes les propriétés (y compris elle-même). Un tel système est dit *imprédicatif*. Le système F de Girard est obtenu en ajoutant la quantification sur les propriétés à la logique minimale. Il forme ainsi le système d'expressivité minimale où les raisonnements imprédicatifs sont permis.

L'imprédicativité est extrêmement expressive. En système F, les structures de données n'ont pas besoin d'être définies. Elles sont toutes encodables au moyen de quantifications et d'implications (voir les travaux de Böhm etBerarducci [9]).

La puissance de l'imprédicativité cause des suspicions chez certains constructivistes quant à son admissibilité. Par contre, l'idée de ne pas se cantonner à un nombre fini de structures

de données a été réexploitée.

De même, exprimer des propriétés qui varient selon la valeur des données dont elles dépendent s'est révélée une brique élémentaire des systèmes logiques constructifs. Le *logical framework* (LF) [48] ou $\lambda\pi$ [20] est le langage générique pour raisonner sur ces systèmes.

La généralisation : les inductifs

Martin-Löf a proposé un système logique prédicatif disposant des structures de données de la logique intuitionniste et de quantifications sur les valeurs.

Pourtant, toutes les structures de données vues jusqu'ici ont toutes un canevas similaire. Elles sont engendrées par un nombre fini de *constructeurs* et les utiliser revient à énumérer le comportement à adopter dans le cas de chacun des constructeurs. De plus, prouver une propriété dans les cas récursifs peut se faire grâce à cette même propriété sur les sous-parties récursives.

De cette constatation, Dybjer [22] a proposé une généralisation de la théorie des types de Martin-Löf avec un monde ouvert. Toute structure de données qui respecte un canevas assurant sa bonne formation est définissable puis utilisable directement.

Les systèmes logiques sont d'une importance fondamentale pour l'informatique. Ces recherches sont en réalité développées non pas seules mais en interaction avec l'informatique grâce à deux découvertes fondamentales du vingtième siècle. Pour poursuivre continûment le déroulé de cette introduction, elles sont présentées ici dans l'ordre antichronologique.

Correspondance preuves/programmes

Pour tenter d'assurer la correction des programmes, on a cherché à les classifier. Pour cela, on leur a associé un *type*, une spécification construite par des *règles d'inférence* à partir de la *syntaxe* du programme qui spécifie son comportement. Si l'on sait qu'un morceau de programme est une fonction qui prend un entier et renvoie une liste de vrai ou faux, nous savons qu'il est impossible de l'élever au carré ou de lui donner en entrée un tableau de nombres à virgule.

L'intérêt premier des types est de signaler tôt et systématiquement au programmeur toute une classe d'erreurs qu'il a pu commettre, sans attendre que ce programme plante lors de son exécution. Mais avec leur introduction intervient une révolution. Les types ont donné corps au travail des logiciens constructivistes. Prouver que la propriété que A et B implique A ou B est identique à écrire une fonction qui prend la paire d'un élément de type A et d'un élément de B et qui renvoie un élément soit de type A soit de type B. C'est la *correspondance de Curry-DeBruijn-Howard* ; la logique nourrit l'informatique.

Prouver des théorèmes est isomorphe à écrire des programmes si l'on est capable de donner un *contenu calculatoire* aux axiomes et aux constructeurs de notre logique. Le coeur de ce dispositif est la correspondance entre la fonction et l'implication logique. Autrement dit la logique minimale a pour langage correspondant le λ-calcul simplement typé.

Le bloc de données et la conjonction de propriétés sont les deux facettes du même objet. Le choix et disjonction aussi.

Derrière la récurrence, il y a un programme clair : les fonctions récursives, c'est-à-dire celles qui se rappellent elles-même. L'arithmétique de Péano a donc un langage de programmation correspondant dont le comportement calculatoire est basé sur le système T de Gödel.

Burstall propose avec Hope [15] un langage de programmation ayant un mécanisme de *filtrage* pour raisonner par cas et la possibilité de définir des types algébriques. Cette idée a mené à la définition de la famille de langages fonctionnels ML ainsi qu'à Haskell [31] et a donné le support nécessaire à la définition de la théorie des types de Martin-Löf.

Machines universelles

Avant la découverte de cette correspondance entre calculs qui terminent et preuves de théorème, les différentes manières de calculer avait été prouvées équivalentes.

- Alan Turing cherchant à modéliser le comportement mécanique du mathématicien en le représentant comme une machine munie d'états d'esprits, d'un alphabet, d'un crayon, d'une gomme, de règles pour changer d'état d'esprit et agir localement sur le papier à partir de son état d'esprit et de ce qu'il lit et d'infiniment de feuilles de papiers ;
- John von Neumann réalisant concrètement une machine capable de réaliser quelques opérations de lecture et d'écriture dans un dispositif de stockage d'informations qu'il nomme *mémoire*. Cette machine lisant quelles opérations elle doit faire à partir de sa mémoire ;
- Alonzo Church construisant le λ-calcul, un langage élémentaire pour parler des preuves mathématiques ;

peuvent tous trois calculer les mêmes résultats !

Les preuves qu'un mathématicien peut écrire, il peut les manipuler mécaniquement. Ce qu'il peut faire mécaniquement en concevant un programme, une machine peut le faire. Voici la révolution qui créa l'informatique théorique : raisonner, c'est effectivement programmer, l'informatique peut nourrir la logique.

Avec ces formalismes, il est facile de montrer que ces machines ne peuvent pas tout. Elles ne peuvent par exemple pas prédire dans tous les cas si un algorithme donné s'arrête après un nombre fini d'étapes de calcul. Cette question est plus élémentaire que savoir si deux algorithmes ont les même sorties pour les même entrées. Il n'y a aucune chance de répondre en toute généralité au problème de l'équivalence de programmes.

Programmation fonctionnelle avec types algébriques

La théorie des types de Martin-Löf généralisée propose sur le versant informatique un langage avec un monde ouvert où l'utilisateur peut introduire de nouvelles structures de données à la condition qu'elles soient correctement formées. Ces données sont ensuite utilisables grâce à des fonctions récursives ou du raisonnement par induction. On parle naturellement de *structure de données inductives*.

Pour l'utilisateur, travailler avec des structures de données qu'il a définies exactement pour son besoin lui permet d'écrire des programmes suivant son intuition sans se soucier d'encodage. L'étude mathématique justifie la correction des programmes typés dans ce formalisme.

Ce paradigme de programmation permet d'utiliser de manière sûre et simple les primitives des processeurs.

Réaliser un *choix* est une des opérations élémentaires mises à la disposition du programmeur. Cette action consiste à exécuter une partie ou une autre d'un programme suivant la valeur d'une entrée. De manière primitive dans un processeur, l'opération de branchement conditionnel saute à une instruction plus lointaine en fonction du résultat d'une comparaison d'entiers.

Historiquement, les langages impératifs (C, Java, …) proposent un opérateur de sélection de cas à partir de cette primitive. La commande `switch n` inspecte la valeur de l'entier n puis exécute le code situé entre `case k:` et `break;` si $n = k$. Un comportement par défaut, pour les k n'apparaissant pas, est spécifié grâce à `default:`.

Si un ordinateur ne manipule que des séquence de bits, ce n'est pas le cas de l'intuition humaine. Il est donc communément possible d'associer un nom à un nombre par `#define FLEUR 12` afin de clarifier les programmes en écrivant `switch (plante) { case FLEUR: cueillir; break; default: arroser}`. Par contre, il n'existe aucune garantie sur l'exhaustivité ou la cohérence des choix donnés. Le programme `switch (animal) { case FLEUR: promener; break; default: carresser}` se révélera aberrant à l'exécution mais il est accepté par le logiciel chargé d'en permettre l'exécution : l'interpréteur ou le compilateur.

De même, réserver un bloc de mémoire est primitif. L'accès aux champs d'un bloc dans les langages impératifs n'a par contre aucune garantie. Rien n'empèche donc de lire une mauvaise zone de mémoire. Ces problèmes sont pris en charge par les disciplines de typage.

Types riches

Les systèmes de type qui permettent le plus de contrôle sur les programmes autorisent les types à dépendre de valeurs. Ce gain entraîne deux complications : l'apparition d'annotation de type dans les termes et l'apparition de calculs dans les types (l'impossibilité de ne considérer que l'égalité syntaxique de types).

Le modèle où l'utilisateur donne tout d'abord intégralement un programme ou une preuve, puis la machine le/la vérifie dans un second temps n'est plus satisfaisant. Des annotations sont fastidieuses à écrire alors qu'il est possible d'écrire une fois pour toutes un programme permettant à l'ordinateur de les deviner. De longs calculs que l'utilisateur ne veut pas faire de tête sont parfois nécessaire pour connaître le type que doit avoir un terme. La machine et son utilisateur doivent interagir afin de construire et vérifier incrémentalement les termes. Les *assistants de preuve* sont des logiciels permettant de construire interactivement preuves ou programmes. Ce domaine de recherche prolifique a généré de nombreuses tentatives aux objectifs multiples.

Prouver des théorèmes avec des programmes simplement typés est par exemple permis par Isabelle [54], HOL-light [27] ou ACL2 [32]. On retrouve également des systèmes où la programmation peut être aussi dépendante que la preuve, par exemple Lego [51], Matita [1]

ou `Coq`. Cayenne [3] puis Epigram [39] et Agda [14] offrent par exemple des systèmes tournés en premier lieu vers la programmation. Twelf [49] (ou Dedukti [13]) tentent d'extraire l'universalité des systèmes logiques afin de générer d'autres systèmes.

Ces systèmes offrent à l'utilisateur un langage de surface qui lui évite d'écrire exhaustivement les termes. L'interactivité minimale est une présentation par la machine du type du terme attendu en un point donné. Le choix de la forme à présenter est cruciale pour la compréhension humaine.

Utilisation des données inductives

Dans la différentiation du langage de surface et de celui sur laquelle s'appuie la logique, ce manuscrit distingue des concepts qui sont confondus dans la littérature. Nous employerons des mots synonymes dans des sens distincts car notre propos est précisément de montrer les distinctions et les passerelles des uns aux autres.

Intuitivement et en anticipant légèrement sur la suite qui définit plus formellement chacun des termes, nous distinguons :

- l'analyse de cas élémentaire, cette réécriture du principe d'élimination d'un inductif qui donne le comportement pour chacun des constructeurs et que l'on peut qualifier de filtrage atomique.
- le typage de l'analyse de cas, justement par étude de cas, qui assure que pour chacun des constructeurs, une réponse bien typée est apportée. Cette primitive est identique pour tous les éléments d'un type inductif.
- le filtrage complexe, qui correspond au filtrage généralisé, celui que l'on retrouve dans les langages fonctionnels. Son exécution nécessite alors une phase de compilation vers une forme plus élémentaire.
- et le typage par analyse de couverture, celui qui travaille sur une instance du type inductif spécifiquement et permet de typer un filtrage (généralisé) directement.

Dans le λ-calcul simplement typé ou paramétrique, les deux typages sont équivalents car tous les constructeurs ont le même type. Il n'y a donc pas réellement besoin de s'attarder sur la version élémentaire de l'opération qui n'est qu'un point de passage au cours de la compilation.

Avec des types dépendants, la confusion a perduré alors que le filtrage (l'opération non atomique) n'est pas typable directement dans un système comme le notre qui n'a à disposition qu'un principe d'élimination générique. Qui plus est, l'opération complexe considérée primitive apporte avec elle un axiome supplémentaire (voir 1.7).

Ce manuscrit parlera donc d'analyse de cas (élémentaire) et de filtrage (généralisé) en omettant les qualificatifs. Sa première contribution originale est de proposer une nouvelle compilation du second dans le premier.

Cette première partie du manuscrit est décomposée en trois chapitres. Dans un premier temps, l'algorithme pour atomiser un filtrage complexe sans considération de typage est rappelé. Dans un deuxième temps, le système de typage direct du filtrage complexe tel qu'imaginé par Coquand est décrit. Nous présenterons le plongement que McBride a proposé dans les CCI en utilisant un axiome supplémentaire. Le chapitre 4 de ce manuscrit décrit une com-

pilation d'un sous-ensemble significatif des filtrages dans le CCI pur. C'est la continuité du travail réalisé par Herbelin et Cornes [18].

Réduction des points fixes

Au lieu de définir les récurseurs des types inductifs primitivement, un mécanisme générique de point fixe est proposé. Bien sûr, il faut s'assurer que les points fixes ne peuvent pas produire de calculs infinis. Avec un vérificateur de terminaison, cette formulation est équivalente aux principes de récurrence [23].

Ces points fixes sont définissables par le biais de constantes globales autorisées à s'appeler elles-mêmes ou par des constructions locales aux systèmes. La seconde contribution originale de ce manuscrit est de simuler au niveau de l'utilisateur le premier système dans le second. La description de se mécanisme constitue le chapitre 5.

Le vérificateur de terminaison doit pouvoir interagir au mieux avec la volonté de l'utilisateur. Par exemple, il doit permettre la réutilisation de code. Il doit aussi ne pas être perturbé par les constructions dont le système a besoin pour assurer le typage du terme. La troisième contribution originale de ce manuscrit est un algorithme présenté chapitre 6 qui va dans ce sens. Il se base sur un critère syntaxique de décroissance structurelle.

1 Un langage fonctionnel avec types riches

Le langage formel utilisé dans ce manuscrit est un λ-calcul à la Church avec des types algébriques et des constantes globales écrit en déduction naturelle. Il va être décrit progressivement dans ce chapitre.

1.1 Conventions d'écriture

Des *listes* sont utilisées tout au long de ce manuscrit. L'écriture adoptée pour représenter la liste de *n* "x" est $x_1 \cdots x_n$. Par convention, l'indice s dans $x_1 \cdots x_s$ est utilisé quand le nombre d'éléments n'importe pas.

Les *déclarations* sont notées $(x : T)$ où T est le type de la variable x. Les listes de déclarations $(x : X)\ (y : Y)\ (z : Z) \ldots$ dans lesquels x apparaît dans Y, x et y apparaissent dans Z, … seront intensivement utilisées. Elles sont appelées des *télescopes* et disposent d'une notation spécifique : $\overrightarrow{(a_i : A_i)}$ désigne $(a_1 : A_1) \cdots (a_i : A_i)$.

Remarque Dans l'assistant de preuve `Coq`, des *définitions* (`let` $z : Z := t$ `in` u) peuvent entrecroiser des déclarations au sein des télescopes pour obtenir $(x : X)\ (y : Y)\ (z : Z := t) \ldots$. Il est néanmoins toujours possible de remplacer les variables définies par leur définition et de traiter des listes de déclarations uniquement. Ceci simplifie la présentation.

Le langage défini ici se traduit symbole pour symbole en une structure de données utilisables en pratique pour une implantation [1], mis à part la représentation des variables. Une machine utilise des indices de Bruijn pour avoir une représentation canonique des variables, ce qui évite les problèmes de capture. Ce manuscrit utilise par souci de lisibilité des variables nommées en respectant les conventions usuelles. La convention de Barendregt évite les problèmes de capture en imposant des noms de variable tous distincts et l'α-équivalence identifie les termes égaux à renommage des variables liées près.

La syntaxe utilisée est une réminiscence de celle des versions 8 de `Coq`.

Lors de définitions abstraites, les couleurs et les noms tenteront de faciliter la lecture. Il sera par exemple question de termes t, de types T, de constantes globales c et de variables x.

1. Elle est exactement déduite de la structure `Constr.t` du code de `Coq` v8.5.

1.2 Le cœur fonctionnel

1.2.1 Syntaxe

Le λ-*calcul* est un modèle de calcul qui permet de manipuler aisément des lieurs. Tous les programmes dans ce modèle sont construits à partir de fonctions. Le mot *terme* est communément employé pour désigner un de ces programmes.

Il existe plusieurs formulations du λ-calcul. Ce manuscrit utilise une variante typée : le λ-calcul à la Church. Le but du typage est ici d'interdire les calculs infinis. Certaines annotations de type sont explicites afin que le typage reste décidable malgré des types très expressifs.

Stefano Berardi et Jan Terlouw ont proposé un cadre très générique pour étudier les systèmes de type : les systèmes de types purs. Ce cadre est une généralisation du λ-cube de Barendregt ([4]) où chaque dimension représente une possibilité de dépendance en plus et chaque sommet un système de type connu.

Le système de type utilisé ici est le sommet du λ-cube. Termes et types partagent la même syntaxe afin de dépendre les uns des autres. Il a été introduit par Thierry Coquand [16] sous le nom de *calcul des constructions*. Sa syntaxe est donnée figure 1.1

$$u, t, S, T ::= x \mid \lambda(x : T) \Rightarrow t \mid t\ u \mid \forall(x : S), T \mid \mathrm{Set} \mid \mathrm{Type}$$

FIGURE 1.1 – Grammaire du cœur fonctionnel (CoC)

L'*abstraction* de la variable x de type T dans le terme t s'écrit $\lambda(x : T) \Rightarrow t$. Par souci de concision, $\lambda(x : X) \Rightarrow \lambda(y : Y) \Rightarrow t$ est noté $\lambda(x : X)\ (y : Y) \Rightarrow t$ et $\lambda(x : X)\ (y : X) \Rightarrow t$ est noté $\lambda(x\ y : X) \Rightarrow t$.

L'*application* du terme u au terme t est noté $t\ u$ sans symbole explicite. Elle est conventionnellement associative à gauche.

Au niveau des types, le *produit* jouit des même facilités d'écriture que l'abstraction. Le terme $\forall(x\ y : S)\ (z : T), s$ désigne $\forall(x : S), \forall(y : S), \forall(z : T), s$.

Deux constantes, les *sortes* Set et Type sont aussi données.

Le caractère _ est utilisé pour désigner une variable liée qui n'apparaît pas dans le corps du lieur. Ainsi dans le cas d'une abstraction sur une variable non utilisée, on écrit $\lambda(_ : T) \Rightarrow t$.

Dans le cas du produit, $A \to B$ est une notation pour $\forall(_ : A), B$.

1.2.2 Evaluation

Le calcul intervient lorsqu'un argument est appliqué à une abstraction. Une telle situation est appelée un *rédex*. Le terme se réduit alors vers le corps de la fonction dans lequel l'argument remplace la variable de l'abstraction. Cette opération s'appelle une β-*contraction* et se note formellement

$$(\lambda(x : T) \Rightarrow t)\ u \quad \mapsto_\beta \quad t[u/x]$$

La notation $t[u/x]$ désigne la substitution de la variable x par le terme u dans le terme t.

La clôture par congruence [2], réflexivité et transitivité de cette réécriture est appelée *réduction*. Dans la suite, $\mapsto_\beta$ ferra référence à la réduction.

Deux termes t et t' sont convertibles (noté $t \equiv t'$) s'ils se réduisent sur un même terme : $t \mapsto_\beta v$ et $t' \mapsto_\beta v$.

L'ordre dans lequel sont réalisées les réécritures n'a pas d'incidence sur la convertibilité. La réduction est confluante : quelle que soit la manière dont les réécritures sont réalisées à partir d'un terme, les termes obtenus restent toujours convertibles.

En définissant une réduction parallèle qui réalise la réécriture sur toutes les partie du terme en une seule étape, on obtient une réduction que l'on peut simuler à l'aide des règles définies au-dessus. Cette réduction parallèle réduit les deux termes issus d'une paire critique sur le même terme en une étape (propriété du diamant).

Les stratégies de réduction (discutées chapitre 5) n'auront donc pas d'effet sur l'expressivité.

L'autre point important de la réduction est sa *terminaison*. Une *valeur* ou forme normale est un terme sur lequel aucune réécriture β n'est applicable sur aucune partie. Il est indispensable que tous les termes du langage se réduisent sur une valeur (unique par conséquence de la confluence). Le typage l'impose. La preuve passe par la construction d'un modèle et a fait en particulier l'objet de travaux par Coquand [16], Werner - Miquel [42], Lee [35], Barras [5], …

1.2.3 Typage

Le système de type du calcul des constructions est présenté figure 1.2. La sorte s désigne indifféremment Set ou Type.

Il est décrit sous la forme de *jugements* qui donne le type d'un terme à partir du type des variables apparaissant dans ce terme. Le type des variables est stocké dans un *environnement* (conventionnement nommé Γ) qui est soit vide ($\emptyset$) soit composé de la déclaration de la variable x de type T suivi du sous contexte Γ ($\Gamma, (x : T)$).

1.3 Déclarations globales

1.3.1 Syntaxe

L'utilisateur n'écrit pas un terme monolithique mais une succession de *définitions globales* notées ($c : T := t$) qui forment un environnement Δ. Par réminiscence de la syntaxe de Coq, la syntaxe pour définir la constante c dans les exemples de ce manuscrit est
`Definition c : T := t.`

Il est aussi possible de déclarer des *axiomes*. Ce sont des constantes sans valeur notée ($x : T$).

Pour éviter les répétitions, `Definition c : ∀(x : S), T := λ(x : S) ⇒ t.` peut être écrit `Definition c (x : S) : T := t.`

2. Par congruence, nous entendons les règles de la forme $t \mapsto_\beta t'$ et $T \mapsto_\beta T'$ implique $\lambda(x : T) \Rightarrow t \mapsto_\beta \lambda(x : T') \Rightarrow t'$.

$$\boxed{\Gamma \vdash t \in T}$$

$$\frac{}{\Gamma, (x : T) \vdash x \in T} \qquad \frac{\Gamma \vdash S \in s \qquad \Gamma \vdash y \in T}{\Gamma, (x : S) \vdash y \in T}$$

$$\frac{\Gamma, (x : S) \vdash t \in T \qquad \Gamma \vdash \forall (x : S), T \in s}{\Gamma \vdash \lambda (x : S) \Rightarrow t \in \forall (x : S), T} \qquad \frac{}{\Gamma \vdash \mathrm{Set} \in \mathrm{Type}}$$

$$\frac{\Gamma \vdash t \in \forall (x : S), T \qquad \Gamma \vdash u \in S \qquad T[u/x] \equiv T' \qquad \Gamma \vdash T' \in s}{\Gamma \vdash t\ u \in T'}$$

$$\frac{\Gamma \vdash S \in \mathrm{Set} \qquad \Gamma, (x : S) \vdash T \in \mathrm{Set}}{\Gamma \vdash \forall (x : S), T \in \mathrm{Set}} \qquad \frac{\Gamma \vdash S \in s \qquad \Gamma, (x : S) \vdash T \in \mathrm{Type}}{\Gamma \vdash \forall (x : S), T \in \mathrm{Type}}$$

Figure 1.2 – Typage du calcul des constructions

Les termes peuvent faire référence à des constantes globales précédemment définies. Une constante est un constructeur de terme (figure 1.3).

$$u, t, S, T ::= x \mid \lambda (x : T) \Rightarrow t \mid t\ u \mid \forall (x : S), T \mid \mathrm{Set} \mid \mathrm{Type} \mid c$$

Figure 1.3 – Grammaire de CoC avec définition

1.3.2 Evaluation

Au cours d'un calcul, une constante est remplacée par le corps de sa définition donnée par l'environnement global (communément appelé Δ). Cette opération est appelée δ-expansion et se note :

$$\Delta, (c : T := t), \Delta' \vdash c \quad \mapsto_\delta \quad t$$

La réduction intègre maintenant cette règle de réécriture et est notée $\Delta \vdash \quad \mapsto_{\beta\delta} \quad$. La confluence reste valable. La réduction parallèle conserve la propriété du diamant. Le dépliage de constante n'a pas d'incidence sur la terminaison.

1.3.3 Typage

Le jugement de typage est maintenant paramétré par l'environnement des constantes définies. Le type d'une constante est donné par cet environnement.

$$\frac{}{\Delta, (c : T := t), \Delta' ; \Gamma \vdash c \in T} \qquad \frac{}{\Delta, (c : T), \Delta' ; \Gamma \vdash c \in T}$$

.

L'environnement global doit être bien formé, c'est-à-dire vide ou obtenu par ajout de constantes aux noms frais et bien typées.

$$\boxed{\Delta \dashv}$$

$$\frac{}{\emptyset \dashv} \qquad \frac{\Delta \dashv \quad \Delta ;\emptyset \vdash T \in s \quad \Delta ;\emptyset \vdash t \in T \quad c \notin \Delta}{\Delta, (c : T := t) \dashv}$$

$$\frac{\Delta \dashv \quad \Delta ;\emptyset \vdash T \in s \quad c \notin \Delta}{\Delta, (c : T) \dashv}$$

FIGURE 1.4 – Environnements bien formés

1.4 Types de données algébriques strictement positifs

1.4.1 Syntaxe

Le langage considéré jusqu'à maintenant permet de définir des structures de données par codage imprédicatif.

Néanmoins, comme suggéré dans l'introduction, manipuler des données de manière primitive est plus naturel. Les constructions utilisées pour rendre cela possible sont les structures *inductives*. L'ajout des types inductifs primitifs décrit ci dessous donne le *Calcul des Constructions Inductives* dû à Christine Paulin et Thierry Coquand [17].

Les structures de données sont définies uniquement au sein de l'environnement global. La syntaxe des contextes globaux est enrichie de l'entrée $\left\{ \begin{array}{l} (\mathbf{C_1} : T_1) \\ \cdots \\ (\mathbf{C_i} : T_i) \end{array} \quad @ \quad \begin{array}{c} \overrightarrow{(a_m : A_m)} \to \\ \mathbf{I} : T \end{array} \right\}$.

Les exemples du manuscrit ainsi que la syntaxe de `Coq` v8 utilisent une syntaxe différente plus lisible où le bloc ci-dessus est écrit `Inductive` $\mathbf{I}\ \overrightarrow{(a_m : A_m)}$ `:` T `:=` $|(\mathbf{C_1} : T_1) \cdots |(\mathbf{C_i} : T_i)$`.`

La figure 1.5 montre une déclarations d'inductifs écrite en forme normale afin d'introduire la terminologie employée pour les manipuler.

`Inductive` $\mathbf{I}\ \overrightarrow{(a_m : A_m)}$ `:` $\forall \overrightarrow{(b_n : B_n)}$`,` Set `:=`
$|\ \mathbf{C_1} : \forall \overrightarrow{(x_{1s} : T_{1s})}$`,` $\mathbf{I}\ a_1 \cdots a_m\ t_{11} \cdots t_{1n}$
...
$|\ \mathbf{C_i} : \forall \overrightarrow{(x_{is} : T_{is})}$`,` $\mathbf{I}\ a_1 \cdots a_m\ t_{i1} \cdots t_{in}$`.`

FIGURE 1.5 – Squelette d'un inductif du CCI

$C_1 \cdots C_s$ sont les *constructeurs* du *type inductif* $\mathbf{I}$.

$a_1 \cdots a_m$ sont les *paramètres* de l'inductif.
$b_1 \cdots b_n$ forment la *signature d'index* de l'inductif.
$x_1 \cdots x_s$ sont les *arguments* du constructeur **C**.
$t_1 \cdots t_n$ sont les *index du constructeur* **C**.

Par construction,

— Le type des constructeurs doit être en forme constructeur par rapport à **I**. La syntaxe des *forme constructeur par rapport à X* est Co : := $X\ t_1 \cdots t_s \mid \forall(x : T)$, Co avec $T = X\ u_1 \cdots u_s$ ou X n'apparaît dans T et X n'apparaît pas dans les termes t et $u_1 \cdots u_s$. L'occurrence finale de X est la *conclusion du type du constructeur.*

— Les paramètres sont des constantes d'une définition. Il doivent rester identiques dans la conclusion du type des constructeurs et dans la définition de l'inductif.

Un exemple d'inductif est la définition de l'égalité donnée figure 1.6.

```
Inductive eq (A : Set) (a : A) : A → Set := | eq_refl : eq A a a.
```

Lorsque l'on écrit **eq** T t u, la valeur du type T est imposée par les termes t et u. Par conséquent, lorsque le type T n'est pas le point clé de l'explication, **eq** T t u est noté par la suite t = u.

Figure 1.6 – Définition de l'égalité

Une fois définis, inductifs et constructeurs sont utilisables dans les termes comme des constantes. Les constructeurs ont un statut particulier. Deux constructeurs sont prouvablement distincts et l'ensemble des habitants d'un type inductif est statiquement connu par l'environnement global. Grâce à cela, Christine Paulin a défini une règle d'élimination des données par *analyse de cas.* La syntaxe employée est donnée figure 1.7.

$\texttt{case}\ t\ \texttt{predicate}\ P\ \texttt{of} \mid br_1 \cdots \mid br_i\ \texttt{end}$

Figure 1.7 – Analyse par cas sur t de type $\mathbf{I}\ u_1 \cdots u_m\ v_1 \cdots v_n$

Le terme t inspecté par l'analyse de cas est appelé *terme inspecté.* L'annotation de typage P est appelée *clause de retour* ou *prédicat de retour.* Le terme br_j appelé *branche du constructeur* C_j correspond au comportement du programme si le terme inspecté débute par le *j*-ième constructeur. Afin de fournir des noms pour représenter les arguments du constructeurs, une branche débute par autant d'abstractions que le constructeur lui correspondant a d'arguments.

Le langage `Gallina` fournit une primitive plus haut niveau qui sera l'objet d'étude des chapitres suivants. Néanmoins, l'analyse de cas écrite en forme dite η-longue (où il y a un λ pour toutes les abstractions) s'injecte dans la syntaxe de `Gallina` en écrivant

Terme de `Constr.t`

$$\texttt{case}\ t\ \texttt{predicate}\ \lambda\overrightarrow{(b_n : B'_n)}\ (x : \mathbf{I}\ u_1 \cdots u_m\ b_1 \cdots b_n) \Rightarrow T\ \texttt{of}$$
$$|\lambda\overrightarrow{(x_{1s} : T'_{1s})} \Rightarrow br_1 \cdots |\lambda\overrightarrow{(x_{is} : T'_{is})} \Rightarrow br_i$$
$$\texttt{end}$$

où $T'_{jk} := T_{jk}[u_1 \cdots u_m/a_1 \cdots a_m]$ et $B'_k := B_k[u_1 \cdots u_m/a_1 \cdots a_m]$

Terme de Gallina

```
  match t as x in I b1 ··· bn return T with
| C1 x11 ··· x1s => br1
...
| Ci xi1 ··· xis => bri
end
```

L'analyse de cas n'est pas suffisante pour définir des fonctions par analyse de cas récursives. Une notion de *point fixe* est nécessaire. La syntaxe employée est $\mathsf{fix}_i\ (f : T := t)$ où f est une variable libre dans t. L'indice i est appelé l'*argument récursif* du point fixe. Il désigne le numéro de l'argument à inspecter pour savoir s'il faut déplier le point fixe (comme expliqué section suivante).
`Fixpoint` $f\,(x_1 : S_1) \cdots (x_i : S_i)$ `:` T `:=` t`.` est un alias plus concis pour
`Definition` f `:` $\forall (x_1 : S_1) \cdots (x_i : S_i),\ T$ `:=` $\mathsf{fix}_i\ (f : \forall (x_1 : S_1) \cdots (x_i : S_i),\ T := \lambda (x_1 : S_1) \cdots (x_i : S_i) \Rightarrow t)$`.`

En résumé, la grammaire exhaustive des termes est écrite figure 1.8. Elle est nommée mini-calcul des constructions inductives car le calcul des constructions inductives à proprement parler inclus par exemple une hiérarchie infinie de sortes avec sous-typage ou des structures de données coinductives qui sont hors du champ de l'étude réalisée dans ce manuscrit.

$$u, t, S, T ::= x \mid c \mid \forall (x : S),\ T \mid s \mid \lambda (x : T) \Rightarrow t \mid t\ u \mid C \mid I$$
$$\mid\ \mathtt{case}\ t\ \mathtt{predicate}\ P\ \mathtt{of}\ t_1 \cdots t_s\ \mathtt{end}$$
$$\mid\ \mathsf{fix}_i\ (f : T := t)$$

Figure 1.8 – Grammaire des termes utilisés (mini-CCI)

1.4.2 Evaluation

Une analyse de cas dont le terme inspecté est le constructeur C_j appliqué à des arguments $u_1 \cdots u_s$ se réduit vers la branche de C_j appliquée aux termes $u_1 \cdots u_s$. C'est la ι-contraction.

$$\Delta \vdash \mathtt{case}\ C_i\ u_1 \cdots u_n\ \mathtt{predicate}\ T\ \mathtt{of}\ |t_1 \cdots |t_m\ \mathtt{end} \quad \mapsto_\iota \quad t_i\ u_1 \cdots u_n$$

La dernière règle est la ϕ-expansion. Un point fixe se déplie si son argument récursif commence par un constructeur. Ce conditionnement du dépliage à la forme de l'un des arguments est indispensable. Remplacer systématique la variable représentant le point fixe par le corps du point fixe amène des dépliages infinis lorsque la règle est appliquée sur des sous-termes ayant des variables libres.

$$\Delta \vdash (\mathsf{fix}_i\ (f : T := t))\ u_1 \cdots u_{i\text{-}1}\ (C\ v_1 \cdots v_n) \quad \mapsto_\phi \quad t[\mathsf{fix}_i\ (f : T := t)/f]\ u_1 \cdots u_{i\text{-}1}\ (C\ v_1 \cdots v_n)$$

La réduction enrichie de ces deux règles est notée $\Delta \vdash \quad \mapsto_{\beta\iota\phi\delta}$. Sa confluence est toujours prouvable grâce à une réduction parallèle vérifiant la propriété du diamant. La terminaison demande la construction de nouveaux modèles qui furent par exemple l'objet du travail de Coquand - Paulin [46], Luo [37], Barras [6], ...

1.4.3 Typage

Un type algébrique **I** est un *inductif* si les occurrences de **I** dans les télescopes des arguments des constructeurs sont en *positions strictement positives*. Elles ne doivent jamais apparaître à gauche d'un produit dans un type.

Pour le reste, on étend la règle de bonne formation des environnements globaux (figure 1.4) par la déclaration d'un type inductif :

$$\frac{\Delta \dashv \quad \Delta ;\emptyset \vdash \forall\overrightarrow{(a_m : A_m)},\ T \in \mathrm{Type} \quad \Delta ;(I : \forall\overrightarrow{(a_m : A_m)},\ T), \emptyset \vdash \forall\overrightarrow{(a_m : A_m)},\ T_1 \in \mathrm{Set} \quad \cdots \quad \Delta ;(I : \forall\overrightarrow{(a_m : A_m)},\ T), \emptyset \vdash \forall\overrightarrow{(a_m : A_m)},\ T_i \in \mathrm{Set}}{\Delta, \left\{ \begin{array}{l} (C_1 : T_1) \\ \cdots \\ (C_i : T_i) \end{array} \ @ \ \begin{array}{c} \overrightarrow{(a_m : A_m)} \rightarrow \\ \mathbf{I} : T \end{array} \right\} \dashv}$$

Dans notre restriction mini-CCI, le type de l'inductif (Type) et de ses constructeurs (Set) sont fixes alors qu'en `Coq` ils pourraient varier.

L'extension du typage des termes est donné figure 1.9.

Une fois un inductif déclaré dans l'environnement global, son type et le type de ses constructeurs sont directement donnés par cet environnement.

> Une analyse de cas est une écriture particulière du schéma d'élimination d'un inductif. (Ce schéma correspond à l'encodage imprédicatif d'un inductif [47]).

Lors d'une élimination dépendante, chaque branche attend un type différent. Puisque les types dépendent des termes, le type d'une branche dépend du constructeur auquel il correspond. Le type global de la réponse renvoyée par l'analyse de cas est lui fonction du terme inspecté.

Le typage d'un point fixe n'est pas surprenant. L'originalité de la règle présentée est la condition Decr(Δ,t,f,i). Il correspond à une condition de bonne formation appelée *condition de garde* qui interdit les points fixes divergents. Son étude précise est l'objet du chapitre 6. Son principe général est de garantir que "le nombre de constructeurs en tête de l'argument récursif" est une grandeur strictement décroissante. Cette propriété garantit que seul un nombre fini d'appels récursifs peut avoir lieu si l'argument est un terme clos.

1.5 Des structures de données d'exemple

Ce manuscrit réalise pour chaque partie une présentation incrémentale informelle des problématiques avant d'exposer la réponse formelle apportée. En plus des structures de données ayant des rôles très particuliers en théorie des types comme l'égalité déjà présentée figure 1.6, ces descriptions pédagogiques s'appuient sur des structures de données courantes.

En premier lieu vient la représentation unaire des entiers naturels (figure 1.10) qui fournit une structure de donnée récursive élémentaire.

$$\boxed{\Delta;\Gamma \vdash t \in T}$$

$$\frac{}{\Delta, \left\{\begin{array}{l}(\mathbf{C}_1 : T_1)\\ \cdots \\ (\mathbf{C}_i : T_i)\end{array} @ \begin{array}{c}\overrightarrow{(a_m : A_m)} \to \\ \mathbf{I} : T\end{array}\right\}, \Delta';\Gamma \vdash \mathbf{I} \in \forall\overrightarrow{(a_m : A_m)},\ T}$$

$$\frac{}{\Delta, \left\{\begin{array}{l}(\mathbf{C}_1 : T_1)\\ \cdots \\ (\mathbf{C}_i : T_i)\end{array} @ \begin{array}{c}\overrightarrow{(a_m : A_m)} \to \\ \mathbf{I} : T\end{array}\right\}, \Delta';\Gamma \vdash \mathbf{C}_j \in \forall\overrightarrow{(a_m : A_m)},\ T_j}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash z \in \mathbf{I}\ u_1 \cdots u_m\ v_1 \cdots v_n \\ \Delta := \Delta', \left\{\begin{array}{l}(\mathbf{C}_1 : \forall\overrightarrow{(x_1 : T_1)},\ \mathbf{I}\ a_1 \cdots a_m\ t_{11} \cdots t_{1n})\\ \cdots \\ (\mathbf{C}_i : \forall\overrightarrow{(x_i : T_i)},\ \mathbf{I}\ a_1 \cdots a_m\ t_{i1} \cdots t_{in})\end{array} @ \begin{array}{c}\overrightarrow{(a_m : A_m)} \to \\ \mathbf{I} : \forall\overrightarrow{(b_n : B_n)},\ \mathsf{Set}\end{array}\right\}, \Delta'' \\ B'_k := B_k[u_1 \cdots u_m/a_1 \cdots a_m] \qquad T'_{jk} := T_{jk}[u_1 \cdots u_m/a_1 \cdots a_m] \\ \Delta;\Gamma \vdash P \in \forall\overrightarrow{(b_n : B'_n)}\ (x : \mathbf{I}\ u_1 \cdots u_m\ b_1 \cdots b_n),\ s \\ \Delta;\Gamma \vdash br_k \in \forall\overrightarrow{(x_k : T'_k)},\ P\ t_{k1} \cdots t_{kn}\ (\mathbf{C}_k\ u_1 \cdots u_m\ x_{k1} \cdots x_{ks})\end{array}}{\Delta;\Gamma \vdash \mathtt{case}\ z\ \mathtt{predicate}\ P\ \mathtt{of}\ |\ br_1 \cdots |\ br_i\ \mathtt{end} \in P\ v_1 \cdots v_n\ z}$$

$$\frac{\Delta;\Gamma \vdash T \in s \qquad \Delta;\Gamma, (f : T) \vdash t \in T \qquad \mathrm{Decr}(\Delta,t,f,i)}{\Delta;\Gamma \vdash \mathsf{fix}_i\ (f : T := t) \in T}$$

FIGURE 1.9 – Typage des inductifs du mini-calcul des constructions inductives

```
Inductive nat : Set := | O :nat | S :nat → nat.
```

Figure 1.10 – Les entiers naturels unaires (zéro ou le successeur d'un entier)

Les listes simplement chaînées (figure 1.11) prennent un *paramètre* de type qui indique la nature des éléments qu'elles contiennent. Néanmoins, tous les constructeurs ont le même type. Si t est une liste d'entier, t peut aussi bien être **nil** que **cons** a b. Les deux constructeurs créent des listes d'entiers. Nous insistons ici car cette propriété n'est pas vrai pour les familles inductive ayant un index.

```
Inductive list (A : Set) : Set :=
| nil :list A
| cons :∀(t : A) (q : list A), list A.
```

Figure 1.11 – Type avec paramètre : Les listes simplement chaînées

Le prédicat de parité d'un entier naturel (figure 1.12) fournit une structure de données avec un index donc réellement dépendant. Il sera l'exemple canonique pour faire apparaître des branches impossibles.

```
Inductive even : nat → Set :=
| even_O :even O
| even_SS :∀(n : nat) (_ : even n), even (S (S n)).
```

Figure 1.12 – Prédicat de parité sur les entiers

Des structures plus compliquées ou ad hoc serviront ponctuellement par la suite.

1.6 Digression sur la représentation des constructeurs et des branches

Des abstractions sont utilisées pour représenter les arguments du constructeur dans les branches d'une analyse de cas. Les constructeurs sont vus comme des constantes dont le type est fonctionnel. Ceci n'est pas conventionnel en programmation.

Les langages qui ont le soucis de s'exécuter vite voient un constructeur comme un *bloc* et ces arguments comme des *champs* de ce bloc. L'analyse de cas n'introduit qu'une variable : celle qui représente le terme inspecté. Par contre, les branches disposent d'une primitive supplémentaire `field` i t qui donne le $i$ème champs de t (à supposer qu'il s'agisse bien d'un bloc).

La première solution simplifie l'étude théorique et en particulier l'écriture de la règle de typage, tandis que l'intérêt intrinsèque de la seconde solution est le partage de code. Par exemple, si un utilisateur définit la structure de donnée
`Inductive I : Set := | A :I → I | B :I | C :nat → I.`
puis écrit (en anticipant très légèrement la syntaxe présentée chapitre 2)

`Definition` destA (x : **I**) : **I** := match x with | A y => y | _ => x end.

Les branches de **B** et de **C** devraient être identiques et donc pouvoir pointer vers le même code.

En se souvenant que la machine utilise des indices de de Bruijn pour représenter les variables, c'est effectivement le cas avec

$\lambda_{\mathbf{I}}$. `case` 1 `predicate` $\lambda_{\mathbf{I}}$. **I** `of` | `field` 1 1 | 1 | 1 `end`

mais pas dans le formalisme du calcul des constructions inductives qui donne

$\lambda_{\mathbf{I}}$. `case` 1 `predicate` $\lambda_{\mathbf{I}}$. **I** `of` | $\lambda_{\mathbf{I}}$. 1 | 1 | $\lambda_{\mathbf{nat}}$. 2 `end`

Il est par ailleurs notable qu'une primitive de projection allégerait aussi les termes issus de l'utilisation d'inductifs à un seul constructeur. Il est bien sûr possible de définir les projections de tels objets (par exemple proj1 **nat** **even** x). Néanmoins, la projection définie comme une constante prend en argument les arguments de l'inductif (dans l'exemple nat et even alors qu'une projection primitive (ici `field` 1 x) n'en a pas besoin. Elle peut être typée directement au moyen de la définition de l'inductif au sein de l'environnement global.

1.7 Expressivité de l'analyse de cas

La clause de retour est explicite, elle peut donc être arbitrairement complexe. Elle est la clé de la grande expressivité de l'élimination dépendante.

Une clause de retour débute par des abstractions représentant des généralisations des index du terme éliminé ainsi que le terme éliminé. Par exemple, supposons avoir un inductif **K** qui a un argument et au moins un constructeur **H** de type $\forall$(z : S), **K** z, la clause de retour pour un terme inspecté u de type **K** t a la forme λ(x : S) (y : **K** x) $\Rightarrow$ T. Rien n'oblige le type attendu en retour à utiliser ces abstractions. Dans l'exemple, cela donnerait λ(x : S) (y : **K** x) $\Rightarrow$ P avec t et u apparaissant dans P mais ni x ni y. Néanmoins, le type demandé dans chacune des branches est alors constant (exactement P) et décorrelé des index du constructeur de la branche (z dans le cas de **H**). On ne peut plus raffiner le type grâce aux informations fournies par le constructeur. De plus, les arguments du constructeur sont le plus souvent inutilisables car de types incompatibles. Dans le cas de la branche de **H**, le type attendu est $\forall$(z : S), (λ(x : S) (y : **K** x) $\Rightarrow$ P) z (**H** z). Puisque x et y ne sont pas libres dans P, ce type ce réduit à $\forall$(z : S), P où z n'est pas libre dans P. Il est donc peu crédible de pouvoir utiliser z pour fournir du P.

Pour rendre l'élimination utile, la clause de retour doit utiliser au maximum ces généralisations. Des contraintes de typage que nous allons expliciter limitent cette possibilité.

Le type des abstractions découle uniquement de la définition de l'inductif et non pas du type, plus précis, du terme éliminé. Or, généraliser les termes signifie nécessairement généraliser parallèlement les types. Il existe alors des situations où il est impossible d'écrire un terme de type T k par analyse sur (k : **K** v) car il n'est pas possible de trouver (P : $\forall$(a : A) (b : **K** a), Set) tel que P v k $\equiv$ T k.

Au lieu d'une quantification universelle sur A, il serait souhaitable de pouvoir exprimer une propriété existentielle disant a doit être v. Comme si une analyse de cas dépendante demandait de pouvoir écrire en ce qui concerne des types des motifs où des égalités de variables sont exprimables. La section 3.2 revient sur cette considération.

Une perte d'expressivité découle de cette contrainte. L'égalité de deux termes peut être exprimée comme un type de donnée du langage au moyen de la définition figure 1.6. Elle est ainsi manipulable directement au sein du langage. Il est cependant impossible de définir un terme ayant le type de la figure 1.13 par analyse sur e. La clause de retour qu'il faudrait : ∀(b : A) (e' : **eq** A a b), **eq** (**eq** A a b) e' (**eq_refl** A a) est mal typée car **eq_refl** A a a le type **eq** A a a et non pas **eq** A a b (le type de e').

```
Axiom UIP_refl :
 ∀(A : Set) (a : A) (e : eq A a a), eq (eq A a a) e eq_refl.
```

FIGURE 1.13 – Unicité des preuves d'égalité

Cet exemple n'est pas artificiel et limite la programmation dépendante. Dans [28], Thomas Streicher et Martin Hofmann ont nommé *K* une formulation équivalente (figure 1.14) de cet axiome. Ils ont montré que *K* est indépendant du CCI. Il est en effet admissible mais il existe des modèles qui ne le réalisent pas.

```
Axiom StreitherK (U : Set) :
 ∀(x : U) (P : x = x → Prop), P (eq_refl x) → ∀(p : x = x), P p.
```

FIGURE 1.14 – Axiome *K*

Puisqu'il est compatible avec le calcul des constructions inductives, *K* peut être supposé. Un axiome étant une hypothèse globale qui n'a pas de définition, il n'a jamais la possibilité de se réduire. Calculatoirement, un tel axiome est donc extrêmement préjudiciable à moins d'amener avec lui une forme faible d'*irrelevance des preuves*. En effet, la réduction peut être modifiée pour ignorer la preuve d'égalité t = u qui bloque une réécriture. Si u et t sont convertibles, la réécriture est effacée.

De plus, la section 3.2 montre que *K* est supposé lorsque l'élimination des inductifs est vu comme une couverture de tous les cas possibles. Il a longtemps été considéré comme regrettable qu'introduire *K* de manière externe au CCI bloque toutes les réductions de termes. Une règle d'élimination par cas des inductifs vérifiant *K* a donc été proposée dans [7].

L'apparition de la théorie homotopique des types [52] dans lesquels *K* n'est pas valable mais qui présente d'autres intérêts tempère ce point de vue et justifie le travail présenté chapitre 4.

2 Le filtrage en programmation fonctionnelle

Dans ce manuscrit, nous cherchons à exhiber que la présence de types dépendant rend profondément différents l'analyse de cas élémentaire telle qu'elle est définie dans le CCI et le filtrage généralisé tel qu'il apparaît dans les langages fonctionnels depuis HOPE (le premier langage avec des types algébriques).

L'amalgame entre les deux notions est entretenu car `Coq` offre à l'utilisateur une primitive de filtrage généralisé. En réalité, ce filtrage est *compilé* par le système et cette première partie explique ce procédé de compilation.

Dans les langages avec types algébriques paramétriques (comme par exemple Haskell, OCaml ou SML), il n'y pas besoin d'isoler l'analyse de cas élémentaire du point de vue du typage. Il est néanmoins nécessaire dans le processus de compilation vers un langage machine d'atomiser les tests. Ce travail a été décrit une première fois par Augustsson [2] puis étudié et optimisé de différentes manières, par exemple, par Le Fessant, Maranget[34], Wadler[30, Chapitre 5].

Ce chapitre décrit ces travaux, vu que la structure intermédiaire qu'ils introduisent pour leur compilation est la structure intermédiaire que nous utilisons pour la nôtre.

La compilation proposée par la littérature est optimisante. Des heuristiques permettent de réduire le nombre d'analyse des cas à réaliser pour simuler un filtrage. Ce manuscrit ne discute pas d'optimisation et se contente de décrire la compilation naïve. Néanmoins, il définit la construction optimisante de la structure intermédiaire car il peut ainsi profiter gratuitement du travail sur les optimisations fait dans la littérature.

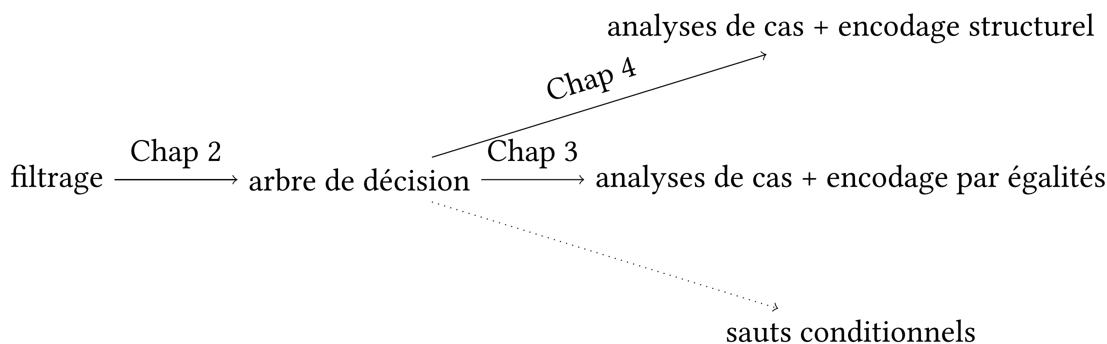


FIGURE 2.1 – Plan de la première partie du manuscrit

Les chapitres suivants expliquent les constructions nécessaire pour aller de la structure intermédiaire qui décrit les choix élémentaires à des analyses de cas du CCI bien typées.

Ces chapitres sont successivement la description de la littérature et l'algorithme original que propose ce manuscrit. Ces chapitres s'occupent seulement de problèmes de types, mais ne reviendront plus sur l'atomisation qui est l'objet de l'étude qui suit.

2.1 Principe du filtrage

Le filtrage (généralisé) n'analyse pas un seul terme mais une liste de termes en parallèle.

Pour choisir le comportement à adopter, il peut inspecter les arguments d'un constructeur puis les arguments d'un constructeur argument d'un constructeur et ainsi de suite. Il n'est pas limité à discriminer selon le premier constructeur en tête du terme.

Le filtrage permet de factoriser le comportement à adopter quand il est identique dans plusieurs cas, il permet par exemple d'écrire un cas par défaut.

Enfin, le filtrage permet de nommer des morceaux des termes inspectés afin de s'en resservir pour décrire le comportement que la suite du calcul doit avoir.

```
Fixpoint beq_half (m n : nat) : bool :=
match m, n with
 | O, O => true
 | S (S i), S j => beq_half i j
 | _ , _ => false
end.
```

Figure 2.2 – Exemple d'inspection en parallèle et en profondeur : n est il la moitié de m ?

Un filtrage est constitué de la liste des termes à inspecter que nous appellerons les termes filtrés et d'une énumération de possibilités qui sont communément appelées les *branches*.

Une branche de filtrage n'est pas composée que d'un terme comme l'est une branche d'analyse de cas. Une branche de filtrage est une paire d'un membre gauche composé d'une liste de *motifs* et d'un membre droit qui est un terme. Chaque branche a autant de motifs que le filtrage a de termes filtrés.

La syntaxe des motifs (figure 2.3) permet d'établir une empreinte pour la tête d'un terme. Si un terme respecte la forme de cette empreinte, il *coïncide* avec le motif.

La sémantique du filtrage est d'exécuter le membre droit de la première branche pour laquelle les membres gauches coïncident avec les termes inspectés en ayant effectué les substitutions nécessaires.

Formellement, la sémantique est défini grâce à

- Une opération $\circ$ de composition de « peut être substitution ». L'absence de substitution ($\perp$) est un élément absorbant.
- une opération de substitution généralisée $[p \leftarrow t]$ décrite figure 2.4 qui prend en argument un motif p et un terme t et renvoie peut être une substitution ; celle à appliquer si le terme coïncide avec le motif.
- une opération $\oplus$ qui sélectionne le premier succès d'une liste de « peut être ».

Elle s'écrit alors

$$\mathbf{match}\ t_1 \cdots t_n\ \mathbf{with}\ |p_{11} \cdots p_{1n} \to u_1 \cdots |p_{s1} \cdots p_{sn} \to u_s\ \mathbf{end} \quad \mapsto \quad \oplus_i(\circ_j[p_{ij} \leftarrow t_j])u_i.$$

Exemple Le terme **cons O** (**cons** (**S** (**S O**)) **nil**) coïncide avec le motif cons _ (cons (S _)) _ mais pas **cons** (**S** (**S O**)) **nil** ou **cons** (**S O**) (**cons O nil**)

$$p ::= \mathbf{C}\ p_1 \cdots p_s \mid _ \mid (\ p1 \mid p2\ ) \mid (\ p \text{ as } x\ )$$

FIGURE 2.3 – Syntaxe des motifs

Le motif "_" est appelé *joker*, n'importe quel terme coïncide avec lui. Un terme coïncide avec le motif **C** $p_1 \cdots p_s$ si, une fois réduit, il est le constructeur **C** appliqué à des arguments coïncidant avec les motifs $p_1 \cdots p_s$. Un terme coïncide avec (p1 | p2) s'il coïncide avec p1 ou avec p2. Enfin, un *alias* (p as x) est un lieur. Il associe le nom x au sous-terme coïncidant avec le motif p. Le motif (_ as x) est noté par simplicité x.

$$\boxed{[p \leftarrow t]}$$

$$[(\ p \text{ as } x\ ) \leftarrow t] \stackrel{\text{def}}{=} [t/x] \circ [p \leftarrow t] \qquad [_ \leftarrow t] \stackrel{\text{def}}{=} \emptyset$$

$$[(\ p \mid q\ ) \leftarrow t] \stackrel{\text{def}}{=} [p \leftarrow t] \oplus [q \leftarrow t]$$

$$[\mathbf{C}_i\ p_1 \cdots p_s \leftarrow \mathbf{C}_i\ t_1 \cdots t_s] \stackrel{\text{def}}{=} \circ_k [p_k \leftarrow t_k] \qquad [\mathbf{C}_i\ p_1 \cdots p_s \leftarrow \mathbf{C}_j\ t_1 \cdots t_s] \stackrel{\text{def}}{=} \bot$$

FIGURE 2.4 – Substitution généralisée pour un motif

Il est notable que les motifs ne peuvent parler que de constructeurs et non pas de n'importe quelle déclaration de l'environnement global. L'avantage de ne traiter que des données à la structure statiquement définie est de connaître l'espace inspecté. Il est par conséquent possible de vérifier la bonne formation des filtrages en contrôlant l'exhaustivité et la non redondance des motifs.

Il existe une autre forme de filtrage, non déterministe cette fois, en programmation logique. La problématique est alors différente et les motifs sont moins contraints. Une branche de filtrage en programmation logique utilise des variables existentielles : "S' il existe x tel que p(x) = t, exécute u" alors que les variables des motifs de la programmation fonctionnelle sont universelles : "pour tout x tel que p(x) = t, exécute u".

Les motifs en programmation logique peuvent être non linéaires : une variable peut apparaître plusieurs fois, signifiant que plusieurs parties d'un terme inspecté doivent être identiques. Il n'est alors plus question de savoir si un terme coïncide avec un motif mais de déterminer les classes d'équivalence auxquelles appartiennent les variables des motifs au sein des termes. Ce calcul est coûteux. Il repose sur un algorithme d'*unification*.

Ce besoin de propriété existentielle au sein du typage de l'élimination dépendante des données a été exhibée section 1.7. C'est pourquoi cette autre forme de filtrage va apparaître section 3.2. Elle est donc mentionnée ici afin de bien la différencier de notre objet d'étude.

2.2 Reconnaitre un motif

Au niveau du processeur, l'unique primitive qui permet le branchement est le saut conditionnel. Reconnaître un constructeur modélisé par un entier ne demande qu'une instruction de comparaison. L'analyse de cas se traduit aisément en sauts conditionnels.

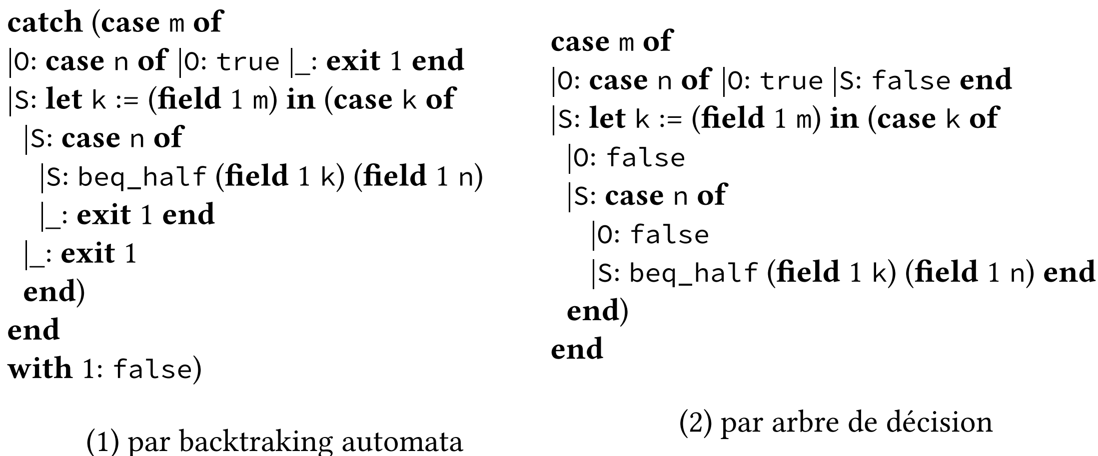
Reconnaître un motif n'est pas aussi immédiat. Pour exécuter un filtrage, la coïncidence avec un motif doit être décomposée en la vérification d'une succession de contraintes élémentaires. Un protocole doit donc être établi afin de compiler le filtrage.

Cette étape de compilation n'est pas nécessaire uniquement à l'exécution mais sert aussi aux contrôles de bonne formation du filtrage. Avec des types de données paramétriques, tous les motifs et toutes les branches ont le même type. Cette partie du typage est réalisable avant compilation. Par contre, contrôler la bonne formation des motifs consiste aussi à s'assurer que tous les cas possibles ont été traité et qu'il n'y a pas de redondance et donc de branches jamais exécutée. Cela revient à exhiber la complétude et la non surcharge de l'arbre des choix à réaliser pour déterminer la branche à suivre, c'est-à-dire travailler exactement sur la structure intermédiaire de la compilation du filtrage.

La littérature propose deux langages intermédiaires pour la compilation : les automates bracktrackant [2, 34] et les arbres de décision [38].

Tous deux utilisent l'analyse de cas comme primitive mais le premier langage maximise la concision du code produit en reposant sur l'usage d'un cas par défaut et d'exceptions alors que le second minimise le nombre de comparaison et s'exprime uniquement par une succession d'analyses de cas telle que présentée section 1.4.

A titre d'exemple, le résultat de la compilation de beq_half (figure 2.2) vers un automate backtrackant est donné figure 2.5.1 et vers un arbre de décision figure 2.5.2. Précisément, la syntaxe utilisée est celle du langage intermédiaire de la compilation d'OCaml `dlambda`.



(1) par backtraking automata

(2) par arbre de décision

FIGURE 2.5 – Compilation de beq_half

Comme elle ne requiert pas d'autres primitives que des analyses de cas, la transformation d'un filtrage vers une succession d'analyses de cas au moyen d'un arbre de décision ouvre la porte au filtrage en calcul des constructions inductives.

Avec des types dépendants, chaque branche a un type différent dépendant du cas considéré. Deux approches existent pour le typage du filtrage.

— La première est un typage avant compilation grâce à une nouvelle théorie logique. Cette approche est décrite dans la section 3.1.
— La seconde est de s'appuyer sur le typage de l'analyse de cas en CCI après compilation du filtrage vers ces derniers. Il faut alors non seulement trouver un arbre de décision mais aussi ajouter les annotations et les constructions nécessaires afin de typer les analyses de cas générées. Ceci revient à trouver un algorithme pour créer des clauses de retour et des termes en tête de branche qui applique les coercions de type nécessaires. Le chapitre 3 décrit un algorithme pour encoder le typage primitif du filtrage que propose la section 3.1. Le chapitre 4 montre une manière alternative et inédite de concevoir des annotations qui ne demandent pas d'étendre la théorie. Ces deux chapitres considéreront qu'un arbre de décision non typé a déjà été construit à partir du filtrage à encoder.

La fin de ce chapitre se concentre justement sur cette étape ne parlant pas de types et donc pas de dépendance : comment transformer un filtrage en un arbre de décision.

2.3 Construire un arbre de décision

2.3.1 Arbre de décision

Formellement, un arbre de décision est un terme dans la grammaire donnée en figure 2.6 agissant sur une pile de termes filtrés.

$$p ::= \mathrm{Tail}\ (t) \mid \mathrm{Leaf}_x :: p \mid \mathrm{Node}_x(p_1 \cdots p_s) \mid \mathrm{Swap}^{i}_{j_1 \cdots j_s} :: p$$

FIGURE 2.6 – Grammaire des arbres de décisions

Tail (t) représente la fin d'un arbre de décision. A condition que la pile des termes à filtrer soit bien vide, t est la réponse apportée.

$\mathrm{Leaf}_x :: p$ est une feuille, c'est-à-dire que le terme en tête de pile n'est pas inspecté plus profondément et p indique comment se comporter pour la queue de la pile. La variable x est l'alias du terme filtré dépilé.

$\mathrm{Node}_x(p_1 \cdots p_s)$ signifie que le terme en tête de pile est inspecté et que le $i$eme p donne le comportement à suivre s'il s'agit du i^e constructeur. Des variables correspondant aux arguments du constructeur sont empilées sur les termes à filtrer dans toutes les branches. Le terme filtré inspecté a pour alias x.

$\mathrm{Swap}^{i}_{j_1 \cdots j_s} :: p$ est une opération d'optimisation. Elle extrude en tête le $i$ème élément de la pile afin de lui appliquer le traitement dicté par p. Cette opération n'est pas rigoureusement identique en présence de types dépendants ou non. D'une liste de termes filtrés réordonnable à souhait, nous passons à la manipulation de télescopes de termes, c'est-à-dire de liste ordonnée du point de vue du typage (les premiers termes apparaissent

dans les types des suivants). Donner la priorité à un terme n'est donc pas uniquement retarder le traitement des autres, c'est aussi annuler le traitement des dépendances du terme choisi. La liste $j_1 \cdots j_s$ donne les numéros des éléments de la liste à jeter car ils sont déterminés par l'analyse du terme sélectionné.

$$\begin{aligned}
\langle \sigma(\mathrm{Tail}\ (\mathrm{t})) \mid \ \rangle &= \sigma(\mathrm{t}) \\
\langle \sigma(\mathrm{Leaf}_{\mathrm{x}} :: \mathrm{p}) \mid \mathrm{t}, \mathrm{q} \rangle &= \langle \{\mathrm{x} := \mathrm{t}, \sigma\}(\mathrm{p}) \mid \mathrm{q} \rangle \\
\left\langle \sigma(\mathrm{Node}_{\mathrm{x}}(\mathrm{p}_1 \cdots \mathrm{p}_s)) \mid \mathbf{C}_{\mathrm{i}}\ \mathrm{t}_1 \cdots \mathrm{t}_{\mathrm{n}}, \mathrm{q} \right\rangle &= \langle \{\mathrm{x} := \mathbf{C}_{\mathrm{i}}\ \mathrm{t}_1 \cdots \mathrm{t}_{\mathrm{n}}, \sigma\}(\mathrm{p}_{\mathrm{i}}) \mid \mathrm{t}_1 \cdots \mathrm{t}_{\mathrm{n}}, \mathrm{q} \rangle \\
\left\langle \sigma(\mathrm{Swap}^{i}_{j_1 \cdots j_s} :: \mathrm{p}) \mid \mathrm{t}_1 \cdots \mathrm{t}_{\mathrm{i}}, \mathrm{q} \right\rangle &= \langle \sigma(\mathrm{p}) \mid \mathrm{t}_{\mathrm{i}}, \mathrm{t}_{\mathrm{k}1} \cdots \mathrm{t}_{\mathrm{ki'}}, \mathrm{q} \rangle \\
\text{avec } \{\mathrm{t}_{\mathrm{k}1} \cdots \mathrm{t}_{\mathrm{ki'}}\} &= \{\mathrm{t}_1 \cdots \mathrm{t}_{\mathrm{i}}\} - \left\{\mathrm{t}_{\mathrm{j}_1} \cdots \mathrm{t}_{\mathrm{j}_s}\right\}
\end{aligned}$$

FIGURE 2.7 – Sémantique des arbres de décisions (σ est une substitution qui donne le terme correspondant à chaque alias)

2.3.2 Mécanique d'atomisation

La liste des branches (appelée *matrice des clauses*) est transformée en un arbre de décision à l'aide d'une fonction qui travaille sur les colonnes de la matrice.

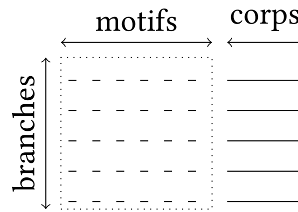


FIGURE 2.8 – Structure d'une matrice de clauses

Les colonnes de la matrice peuvent être considérées comme une pile. Le travail est systématiquement effectué sur la première. L'algorithme présenté ici traite ce cas minimal.

Optimiser la compilation du filtrage, c'est échanger l'ordre des colonnes afin de réduire le nombre d'analyse de cas nécessaires en traitant les colonnes clés d'abord. Cela revient à introduire des constructeurs $\mathrm{Swap}^{i}_{j_1 \cdots j_s} :: \mathrm{p}$ dans l'arbre de décision.

L'algorithme minimal est le suivant :

Quand la matrice n'a plus de colonne de motifs, il doit y avoir exactement une ligne. Moins révèle un filtrage non exhaustif, plus des motifs redondants. La réponse est alors Tail (t) avec t le terme de cette branche restante.

Dans les autres cas, les alias de la première colonne sont unifiés par α-renommage, soit x le nom de l'alias commun.

$$\begin{array}{ll} (\ p_0 \text{ as } x_0\ )\ \text{- - - -} & \text{———} \\ (\ p_1 \text{ as } x_1\ )\ \text{- - - -} & \text{———} \\ (\ p_2 \text{ as } x_2\ )\ \text{- - - -} & \text{———} \end{array} \Longmapsto \begin{array}{lll} p_0\ \text{- - - -} & \text{———} & [x/x_2] \\ p_1\ \text{- - - -} & \text{———} & [x/x_1] \\ p_2\ \text{- - - -} & \text{———} & [x/x_0] \end{array}$$

Ensuite, les disjonctions de la première colonne sont supprimées en dupliquant la ligne avec pour premier motif les deux cas de la disjonction.

$$\boxed{(\,q_1 \mid q_2\,)\ \mathrm{p}_1 \cdots \mathrm{p}_s}\ \ \mathrm{t} \longmapsto \begin{array}{|l|l}q_1\ \mathrm{p}_1 \cdots \mathrm{p}_s & \mathrm{t}\\ q_2\ \mathrm{p}_1 \cdots \mathrm{p}_s & \mathrm{t}\end{array}$$

Enfin, si tous les éléments de la première colonne sont des jokers, la réponse est $\mathrm{Leaf}_{\mathrm{x}} :: \mathrm{p}$ où p est l'arbre de décision correspondant à la matrice privée de sa première colonne.

Sinon, il est temps d'analyser réellement le premier terme filtré. L'arbre de décision débute par $\mathrm{Node}_{\mathrm{x}}(\mathrm{p}_1 \cdots \mathrm{p}_s)$. Pour obtenir la réponse p_i au sous problème associé au constructeur $\mathbf{C}_\mathrm{i}$, on utilise une opération S qui renvoie la nouvelle matrice regroupant les motifs à traiter si le premier terme filtré commence par C_i.

La fonction S $\mathbf{C}_\mathrm{i}$ va transformer la première colonne de motifs en k nouvelles colonnes, avec k le nombre d'arguments de $\mathbf{C}_\mathrm{i}$. Pour cela, elle parcourt les lignes en regardant le motif de tête.

- S'il s'agit du même $\mathbf{C}_\mathrm{i}$ appliqué aux sous motifs $\mathrm{m}_1 \cdots \mathrm{m}_\mathrm{k}$, elle ajoute à la réponse une ligne constitué des $\mathrm{m}_1 \cdots \mathrm{m}_\mathrm{k}$ concaténés au reste de la ligne.

$$\boxed{(\mathbf{C}_\mathrm{i}\ \mathrm{m}_1 \cdots \mathrm{m}_\mathrm{k})\ _\ _\ _\ _}\ \ ____ \longmapsto \boxed{\mathrm{m}_1 \cdots \mathrm{m}_\mathrm{k}\ _\ _\ _\ _}\ \ ____$$

- Si le premier motif commence par un autre constructeur $\mathbf{C}_\mathrm{j}$, la ligne disparaît.

$$\boxed{(\mathbf{C}_\mathrm{j}\ \mathrm{m'}_1 \cdots \mathrm{m'}_{\mathrm{k'}})\ _\ _\ _\ _}\ \ ____ \longmapsto \emptyset$$

- Un joker est transformé en k jokers tout en gardant le reste de la ligne.

$$\boxed{_\ _\ _\ _\ _}\ \ ____ \longmapsto \boxed{_1 \cdots _k\ _\ _\ _\ _}\ \ ____$$

Cette procédure est décrite par Luc Maranget dans [38]. L'objet de son article est de réduire la taille de l'arbre de décision en permutant des colonnes. Il décrit des heuristiques de choix de la colonne à casser en priorité qui permettent d'égaler les performances de la compilation par automates backtrackant en utilisant des arbres de décisions.

Dans un cadre avec types dépendant, l'étude de la correction et de la pertinence des heuristiques reste à traiter, ainsi que la proposition de nouvelles heuristiques spécialisées.

3 Encodage du filtrage dans l'analyse de cas

3.1 Le filtrage dépendant primitif

Il existe des systèmes logiques dans lesquels le filtrage est l'élimination primitive des inductifs. Les langages Agda [45] et Beluga [50] sont basés sur un tel filtrage primitif dont le typage est assuré par analyse de couverture suivant les idées de Thierry Coquand dans [44].

Le principe est de garantir que l'ensemble des motifs membres gauche des branches couvrent l'ensemble des valeurs que peuvent prendre les termes filtrés.

Le typage du filtrage par analyse de couverture présuppose trois classes de propriétés sur les constructeurs d'inductifs.

discrimination Deux constructeurs syntaxiquement distincts sont nécessairement différents. Par exemple, pour les entiers : $\forall (P : \mathrm{Set})\ (n : \mathbf{nat}),\ \mathbf{S}\ n = \mathbf{O} \rightarrow P$

inversibilité Deux instances d'un constructeur sont identiques si et seulement si leurs arguments sont identiques. Pour les entiers : $\forall (m\ n : \mathbf{nat}),\ \mathbf{S}\ m = \mathbf{S}\ n \rightarrow m = n$

acyclicité L'argument d'un constructeur d'inductif ne peut pas être égal au terme entier. Par exemple, pour les entiers naturels, $\forall (P : \mathrm{Set})\ (n : \mathbf{nat}),\ \mathbf{S}\ n = n \rightarrow P$, ce qui est vrai également pour tous les $\mathbf{S}\ (\mathbf{S} \ldots (\mathbf{S}\ n)) = n$.

Lors du typage d'une branche, la correspondance entre les index des termes filtrés et ceux du motif considéré ainsi qu'entre les termes filtrés et le cas considéré se ramène à un problème d'unification ayant pour règles les énoncés ci-dessus. Si t de type $\mathbf{even}\ u$ est éliminé, dans la branche de $\mathbf{even_SS}\ x\ y$ de type $\mathbf{even}\ (\mathbf{S}\ (\mathbf{S}\ x))$, les problèmes d'unification sont $u \stackrel{?}{=} (\mathbf{S}\ (\mathbf{S}\ x))$ au niveau des index et $t \stackrel{?}{=} (\mathbf{even_SS}\ x\ y)$ pour le terme filtré. L'algorithme cherche une substitution des variables apparaissant dans les contraintes telle que les contraintes soient satisfaites. Les variables des motifs peuvent donc être métamorphosées au besoin en des variables existentielles qu'il faut substituer par un terme pour satisfaire les contraintes de type.

— S'il existe une telle substitution, elle est appliquée aux types des variables libres et au type attendu en retour. Elle est aussi appliquée aux variables du motif qui ont été transformées ainsi qu'aux types des variables du motif. Dans l'exemple, le type de y contient x qui pourrait être part de la substitution.

— S'il n'existe aucune substitution satisfaisant le problème, la branche est déclarée impossible et est effacée.

Les variables du contexte sont ici vues comme des variables existentielles et non universelles. Le type UIP_refl, l'axiome indépendant du CIC et équivalent à K défini figure 1.13, est

de ce fait habité par le terme Agda [1]

UIP_refl : ∀ (A : Set) (a : A) (e : **eq** A a a) ↦ **eq** (**eq** A a a) e (**eq_refl** A a)
UIP_refl .B .b (**eq_refl** B b) = **eq_refl** (**eq** B b b) (**eq_refl** B b)

Il est valide car il existe la substitution A := B, a := b telle que la branche **eq_refl** soit bien typée.

Toutes les éliminations ne peuvent pas être réalisées. Le comportement problématique apparaît quand les arguments du terme filtré sont autre chose que des variables et des constructeurs. Un terme ayant pour type (plus x y) = **O** ne peut pas être éliminé directement, l'algorithme d'unification ne sait ni inférer une substitution la plus générale telle que $x + y \equiv 0$ ni décréter qu'il n'y en a aucune et que ce cas est impossible. En effet, l'unification d'ordre supérieur en λ-calcul est indécidable (voir [29]).

Cette restriction est nuancée en Agda en travaillant après réduction et en autorisant des termes quelconques mais syntaxiquement égaux après application de la substitution trouvée par ailleurs. Il est par exemple possible d'éliminer un terme de type (**even** (**S** (**S** **O**))) = (**even** (plus (**S** **O**) (**S** **O**))).

3.2 Encodage du filtrage par des analyses de cas

Le filtrage est une manière concise et facile à relire de présenter des programmes. Cette concision est encore plus vraie avec des types dépendants quand les branches impossibles peuvent être omises. L'exemple systématique pour illustrer ce fait est de pouvoir écrire

even_map2 : ∀ (n : **nat**) ↦ **even** n ↦ **even** n ↦ _
even_map2 .**O** **even_O** **even_O** = ?
even_map2 .(**S** (**S** m)) (**even_SS** .m e1) (**even_SS** m e2) = ?

sans rien dire des cas mixtes **even_O**/**even_SS** impossibles par typage. Il est donc souhaitable d'offrir à l'utilisateur cette syntaxe de surface même si la logique sous-jacente ne traite que d'analyse de cas élémentaire.

Conor McBride et James McKinna ont démontré dans [24] que l'expressivité de l'analyse par cas était celle du filtrage par analyse de couverture si l'on suppose l'axiome UIP de la figure 1.13. La démonstration revient à construire une clause de retour et un entête de branche après avoir obtenu l'arbre de décision pour tous les filtrages typables par analyse de couverture [40].

Cristina Cornes [19] puis Matthieu Sozeau [53] ont mécanisé pour `Coq` la génération de telles clauses de retour et preuves en tête de branche. L'utilisateur dispose de la commande "Program" pour écrire des termes et des tactiques `case_eq`, `inversion` et `dependent_destruction` par échelle croissante d'automatisation dans les preuves. Sur des exemples fortement dépendants, ces techniques sont puissantes mais souffrent des limitations de l'axiome *K* décrites section 1.7.

1. En Agda, .t signifie justement que les contraintes de typage imposent de substituer cette variable de motif par le terme t.

3.2.1 Des types riches exprimant les problèmes d'unification

L'encodage repose sur l'obtention et la manipulation explicite d'égalités dans le langage. Ces égalités correspondent aux problèmes d'unification entre les index des termes filtrés et ceux des constructeurs. Les propriétés précédentes des constructeurs (discrimination, inversibilité et acyclicité) doivent être ensuite bâties explicitement.

Pour obtenir l'information, la solution est d'utiliser le prédicat de retour de l'analyse de cas. Lors de l'élimination d'un entier n pour produire un terme de type T, on rajoute artificiellement une égalité ce qui donne le prédicat de retour λ(m : **nat**) ⇒ m = n → T. L'analyse de cas dans son ensemble a maintenant le type n = n → T, ce qui peut bien redonner T quand on lui applique **eq_refl** **nat** n. L'avantage est que la branche **O** attend maintenant un terme de type **O** = n → T, ce qui fournit l'égalité recherchée. De telles constructions avec la clause de retour jouent un rôle fondamental pour le typage de l'analyse de cas qui sera discuté plus amplement section 3.4.1.

Dans l'exemple précédent, l'égalité ajoutée relie des objets de type fixe (nat). L'encodage complet du filtrage exposé section précédente nécessite de poser des égalités sur des objets dont les types sont eux-même concernés par d'autres égalités. L'égalité simple définie figure 1.6 ne suffit plus, on a alors besoin de définir une *égalité hétérogène* (figure 3.1).

```
Inductive JMeq (A : Set) (a : A) : ∀(B : Set) (b : B), Set :=
 | JMeq_refl :JMeq A a A a.
```

FIGURE 3.1 – Définition de l'égalité hétérogène

Par exemple, la figure 3.2 qui décrit l'inversion d'un double successeur d'un entier a besoin d'égalité hétérogène. En effet, E a le type **even** 2 alors que H a le type **even** m. L'hypothèse 2 = m ne suffit pas à rendre ces deux types convertibles, ce qui empêche de typer E = H.

```
Definition even_inv (E : even 2) : eq (even 2) E (even_SS O even_O) :=
case E predicate λ(m : nat) (H : even m) ⇒ 2 = m → JMeq (even 2) E (even m) H →
   eq (even 2) E (even_SS O even_O) of
  |even_O:λ(e1 : 2 = O) (e2 : JMeq (even 2) E (even O) even_O) ⇒ ...
  |even_SS:λ(n : nat) (E' : even n) ⇒ λ(e1 : 2 = (S (S n)))
    (e2 : JMeq (even 2) E (even (S (S n))) (even_SS n E')) ⇒ ...
end (eq_refl 2) (JMeq_refl (even 2) E).
2 représente (S (S O))
```

FIGURE 3.2 – Exemple d'inversion avec égalité

Même une fois exploitée l'égalité de 2 et m, il est impossible d'exploiter l'égalité hétérogène. Le problème de généralisation mal typée mais indispensable pour écrire une clause de retour apparaît pour réaliser le type de la figure 3.3 (il faudrait pouvoir dissocier les 2 instances de p dans les arguments de la constante rew (formellement définie figure 3.7) pour pouvoir éliminer l'égalité h) exactement comme celui de la figure 1.13. Tous deux sont d'ailleurs logiquement équivalents à *K*.

Une fois muni de cet axiome, l'encodage est mécaniquement réalisable.

```
Axiom eq_rect_eq :
 ∀(U : Set) (p : U) (Q : U → Set) (x : Q p) (h : p = p),
   x = rew U Q p p x h.
```

FIGURE 3.3 – Egalité modulo réécriture de type

3.2.2 Manipulation des égalités d'index

Les règles utilisées par l'unification durant le typage du filtrage par analyse de couverture deviennent des lemmes utilisés pour réaliser des réécritures en tête de branches. Nous allons détailler comment ces lemmes sont construits et utilisés.

Afin d'exprimer une propriété triviale mais qui n'apporte aucune information, on définit la propriété **True** (figure 3.4). Pour exprimer une impossibilité dont tout se déduit, on utilise la propriété **False** (figure 3.5).

```
Inductive True : Set := | I :True.
```

FIGURE 3.4 – La propriété vraie

```
Inductive False : Set :=.
```

FIGURE 3.5 – La propriété fausse

Puisque **False** n'a pas de constructeurs, son schéma d'élimination (figure 3.6) qui sera réutilisé section 4.6 permet toutes les folies.

La figure 3.7 donne le schéma d'élimination de l'égalité. La possibilité qu'il offre d'avoir une propriété pour le terme de droite si on peut la prouver pour celui de gauche est non seulement le moyen de réaliser les réécritures mais aussi une pierre angulaire de l'élaboration des lemmes d'injectivité eux-mêmes.

On peut remarquer qu'il est équivalent d'écrire m = **O** et `case` m `predicate` λ(_ : **nat**) ⇒ Set `of` |**O**:**True** |**S**:λ(_ : **nat**) ⇒ **False** `end`. En effet, si m est effectivement **O** nous n'avons aucune information supplémentaire. En revanche, si m n'est pas **O**, on obtient une preuve de faux qui nous donne la possibilité de fournir une preuve de T pour tout T en utilisant false_rect.

> L'analyse par cas constitue une forme d'égalité pour les termes dans lesquels seuls des variables et des constructeurs apparaissent, ceux visibles comme des *motifs*.

Cette remarque et rew sont des outils suffisants pour discriminer les constructeurs. Pour deux constructeurs distincts C_i et C_j alors le terme de la figure 3.8 a le type **C_j** = **C_i** → **False** [2]. Ce terme permet de conclure dans les cas absurdes.

De manière moins intuitive mais tout aussi systématique, le type ∀(n : **nat**), m = **S** n → T n peut se reformuler `case` m `predicate` λ(_ : **nat**) ⇒ Set `of` |**O**:**True** |**S**:λ(n : **nat**) ⇒

2. ceci est couramment noté $C_i \neq C_j$

```
Definition false_rect (P : Set) (x : False) : P :=
 case x predicate λ(_ : False) ⇒ P of end.
```

Figure 3.6 – ex falso quolibet

```
Definition rew (A : Set) (P : A → Set) (a b : A)
 (p : P a) (e : eq A a b) : P b :=
case e predicate λ(c : A) (e' : eq A a c) ⇒ P c of
|eq_refl:p
end.
```

Figure 3.7 – Brique élémentaire de la réécriture (équivalente au eq_rect de Coq)

T n end. La correspondance entre m et **S** n est assurée. Si m s'avère être autre chose qu'un **S** dans une branche, la branche est une branche impossible. Il suffit de fournir **I** comme preuve triviale (et inutile pour toute exécution).

La preuve d'inversibilité écrite figure 3.9 utilise cette méthodologie.

Pour les preuves d'acyclicité, on utilise des inductions sur la variable apparaissant des 2 cotés. Tous les cas sauf celui du constructeur qui apparaît en tête dans l'égalité cyclique utilisent une discrimination. Ce dernier cas est exactement l'appel récursif après usage d'un lemme d'inversibilité.

3.3 Utilisabilité de l'encodage

Sur la face logique de la correspondance de Curry-deBruijn-Howard, un lemme de mathématique est rarement démontré pour être évalué. Les termes engendrés par une preuve non triviale sont volumineux. Coq propose pour des questions d'efficacité et de modularité de les masquer une fois qu'ils ont été vérifiés. L'utilisateur utilise de fait ses lemmes comme des constantes opaques la majorité du temps. Ceci rend anodin d'utiliser l'axiome K. De plus, les égalités explicitement introduites autorisent un raisonnement équationnel arbitrairement complexe et plus expressif que n'importe quelle théorie décidable. Par exemple, il peut s'agir d'exploiter l'arithmétique de Peano sur des égalités d'entiers impliquant addition et multiplication.

De plus, si aucune égalité hétérogène n'est utilisée et que l'utilisation des égalités fournies dans les branches se limite à instancier des lemmes de discrimination et d'inversibilité similaires à ceux de la section précédente, le comportement calculatoire du filtrage n'est pas altéré.

L'élimination d'un inductif au moyen de cette méthodologie est utile et utilisée. Un algorithme alternatif partiel mais ne nécessitant pas K sera néanmoins proposé dans le chapitre suivant. En effet, d'une part ce manuscrit met l'accent sur l'écriture de programmes et ses problématiques, d'autre part, un tel algorithme représente un support de réflexion sur l'expressivité d'une analyse de cas sans K.

```
Definition C_i_C_j_discr : C_i = C_j → False :=
λ(e : eq … C_j C_i) ⇒
 rew …
  (λ(x : …) ⇒ case x predicate λ(_ : _)⃗ ⇒ Set of |C_j:True |C_s:False end)
  C_i C_j I e.
```

FIGURE 3.8 – Discrimination en action

```
Definition C_i_inj (a b : …) : C_i a = C_i b → a = b :=
λ(e : eq … (C_i a) (C_i b)) ⇒
 rew …
  (λ(x : …) ⇒ case x predicate λ(_ : _)⃗ ⇒ Set of
    |C_i:λ(y : …) ⇒ eq … a y
    |_:λ(_ : _)⃗ ⇒ True end)
  (C_i a) (C_i b) (eq_refl _ a) e.
```

FIGURE 3.9 – Injection en action

3.4 Constructions génériques autour d'une analyse de cas dépendante

3.4.1 Coupures entrelacées

Concevoir intelligemment des clauses de retour implique d'*entrelacer* l'élimination d'un inductif avec des redex de fonction.

Sans considération de typage, il est plus naturel d'écrire le terme (1) que (2) de la figure 3.10. Les deux termes sont néanmoins équivalents et on parle de coupure *commutative* car il serait possible de réduire (2) vers (1).

```
Fixpoint plus (m n : nat) : nat :=
case n
 predicate λ(_ : nat) ⇒ nat of
 |O:m
 |S:λ(n' : nat) ⇒ S (plus m n')
end.
```

(1)

```
Fixpoint plus (m n : nat) : nat :=
case n
 predicate λ(_ : nat) ⇒ nat → nat of
 |O:λ(m' : nat) ⇒ m'
 |S:λ(n' m' : nat) ⇒ S (plus m' n')
end m.
```

(2)

FIGURE 3.10 – Exemple de coupure commutative

Pour raisonner sur des prédicats de parité d'entier définis figure 1.12, les expressions (1) et (2) de la figure 3.11 sont radicalement différentes. Dans (1), la correspondance de type entre les deux variables est perdue alors que dans (2), le type de e1' est modifié simultanément à e2 dans chaque branche.

En effet, l'analyse par cas ne modifie pas les types des variables libres (comme e1 qui est de type **even** n). Les types dans le contexte et les types des constructeurs sont désynchronisés

```
Definition even_inv (n : nat)
(e1 e2 : even n) : ... :=
case e2

predicate λ(m : nat) (_ : even m) ⇒
... of

|even_0:...

|even_SS:λ(m' : nat)
(e : even (S (S m'))) ⇒ ...
end.
```

(1)

```
Definition even_inv (n : nat)
(e1 e2 : even n) : ... :=
case e2

predicate λ(m : nat) (_ : even m) ⇒ even
m → ... of

|even_0:λ(e1' : even O) ⇒ ...

|even_SS:λ(m' : nat)
(e e1' : even (S (S m'))) ⇒ ...
end e1.
```

(2)

FIGURE 3.11 – Coupure entrelacée sur les prédicats de parité

dans les branches (La première branche indique que e2 est **even_0** de type **even O** mais e1 n'a pas changé de type pour autant). Par contre, en abstrayant une variable, elle passe du contexte à la conclusion. Son type est réécrit dans la clause de retour (e1' est cette copie de e1 au type plus précis). L'utilisation des abstractions permet la conservation des correspondances dans l'analyse par cas.

Sur l'exemple, en inversant l'ordre d'introduction de e1 et e2, la coupure commutative aurait été inutile. L'introduction partielle de e2 uniquement pour laisser e1 dans la conclusion aurait suffi parce qu'alors on peut faire une η-réduction. Cette possibilité n'est pas systématique comme nous le verrons dans la section 4.3.

Introduire une coupure pour obtenir une copie d'un terme dans un type en adéquation avec le type du constructeur de la branche considérée n'est pas utile uniquement pour les variables libres. La mémorisation de la valeur des index du type éliminé se fait de la même manière.

Cette technique est par exemple décrite par Bertot et Castéran dans [11] avec des preuves d'égalité. Pour éliminer (p : **even**(plus (**S** x) **O**)) efficacement, mieux vaut savoir que plus (**S** x) **O** = **O** dans le premier cas (afin d'éliminer ce cas) et plus (**S** x) **O** = **S** (**S** m') dans le second. Nous plaçons pour cela le terme **eq_refl** (plus (**S** x) **O**) comme argument du filtrage et réécrivons son type grâce à la clause de retour λ(m : **nat**) (e : **even** m) ⇒ **eq nat** (plus (**S** x) **O**) m → ... afin d'obtenir exactement les égalités recherchées. Elles peuvent ainsi servir à montrer que la branche est impossible ou à obtenir des correspondances entre les types des arguments du constructeur et ceux du contexte et de la conclusion.

3.4.2 Elimination des branches impossibles

Si un inductif n'a pas d'index, la clause de retour de l'élimination d'un terme de ce type n'a qu'une unique variable, celle qui généralise le terme filtré. Le type attendu dans les branches est le type demandé en retour du filtrage dans lequel le terme éliminé est remplacé par le constructeur de la branche. Sur le cas des listes, la figure 3.12 montre que les termes PN et

PC ont des types raffinés.

```
Definition case_list (A : Set) (P : list A → Set) (PN : P nil)
 (PC : ∀(t : A) (q : list A) , P (cons t q)) (l : list A): P l :=
case l predicate λ(l' : list A) ⇒ P l' of
|nil:PN
|cons:λ(t : A) (q : list A) ⇒ PC t q
end.
```

Figure 3.12 – Principe d'élimination d'une liste

La valeur de l'objet filtré est raffinée dans les branches mais son type n'est pas modifié. Le typage de la clause de retour ne pose pas de difficulté.

Un inductif ayant un ou des arguments comme `Inductive` **I** : ∀(a : A),Set := .... est fondamentalement différent d'un inductif tel que `Inductive` **J** (a : A) : Set := .... n'ayant que des paramètres. Tous les constructeurs de **J** ont en effet le type J a alors que les constructeurs de **I** ont un type I t avec t quelconque et dépendant du constructeur.

Alors que l'élimination de (j : J u) ne peut produire que des constructeurs de type **J** u, l'élimination de (k : I v) peut produire des constructeurs de type **I** t (où t dépend du constructeur). Une clause de retour sur **J** n'a donc pas besoin de dépendre de a. C'est pourquoi il n'y a pas de lieur pour les paramètres. Dans la syntaxe de `Gallina`, ils doivent apparaître comme des _. Une clause de retour sur **I** doit par contre avoir un type de la forme ∀(a : A) (b : I a), Set afin que P v k et P t (C x y) soient simultanément typables.

Le type des branches dépend des index du constructeur en même temps que du constructeur. Le type varie d'un constructeur à l'autre.

L'élimination de k doit avoir une branche pour tous les constructeurs **C** de l'inductif **I**. Il est donc nécessaire de fournir un terme de type P t (C x y) même si t est incompatible avec v et qu'il est impossible que k soit le constructeur **C**. L'abstraction sur l'argument de l'inductif donne un moyen de parler de v dans la clause de retour. Par conséquent, l'incompatibilité de t et v est exprimable dans la clause de retour afin que n'importe quel terme soit admis dans la branche **C**. C'est alors une *branche impossible*.

Si un index débute par un constructeur **CC**, la clause de retour va faire une analyse de cas sur cet index qui revoit le type demandé dans le cas de **CC** et le type choisi pour peupler les branches impossibles dans les autres cas. Nous verrons avec précision à la section 4.3 comment écrire une clause de retour exhibant les incompatibilités. Dans le cadre présenté ici toutes les branches impossibles contiennent une preuve de discrimination.

4 Du filtrage à l’analyse de cas structurellement

L’originalité de ce chapitre est de proposer une traduction des arbres de décision vers des analyses de cas sans utiliser la structure de données **eq** (et **JMeq**). Cette construction réduit le préjudice induit par l’absence de l’axiome K en éliminant des situations où l’encodage de la section 3.2 introduit inutilement des égalités hétérogènes. Elle ne tire pas avantage de l’acyclicyté des types de données. Néanmoins, sans axiome ni preuve par récurrence en tête de branche, le comportement calculatoire n’est jamais entravé. La taille syntaxique des termes est aussi potentiellement moindre et leur typage s’avère plus rapide.

4.1 Squelette d’index

Les constructeurs en tête des index du type des termes inspectés jouent un rôle particulier. Ils sont distingués des autres termes en introduisant la notion de *squelette d’index*. Tout d’abord, nous notons $p(t_1 \cdots t_s)$ le motif p vu comme un terme dans lequel le $i$ème joker est remplacé par le terme t_i. Le squelette du terme u est alors la paire d’un motif p et de termes $t_1 \cdots t_s$ ne débutant pas par un constructeur tels que $u = p(t_1 \cdots t_s)$. Le motif p exprime alors ce sur quoi on peut filtrer. Les termes $t_1 \cdots t_s$ sont dénommés les *feuilles du squelette*.

Les motifs de squelette utilisent une restriction de la syntaxe présentée figure 2.3 puisqu’ils ne contiennent ni alias ni choix.

Par exemple, le squelette du terme **even_SS** (**S** (**S** n)) (**even_SS** n t) est (**even_SS** (**S** (**S** _)) (**even_SS** _ _), [n ;n ;t]).

Deux squelettes sont compatibles si il existe une substitution des jokers de la partie motif de l’un qui permet d’obtenir la partie motif de l’autre.

> Pour tout terme u dans une famille inductive, le squelette d’index du type de u est la liste des squelettes des index de son type.

4.2 Quand les égalités sont inutiles

Considérons l’élimination d’un terme x de type **I** $a_1 \cdots a_s$ afin d’obtenir un terme de type P.

Dans le cas le plus simple, les $a_1 \cdots a_s$ sont des variables deux à deux distinctes et de types identiques à ceux des index de **I** lors de sa définition. Le terme x a dans ce cas un type aussi général que la signature de **I**. Il est alors inutile de recourir à des égalités pour éliminer x. La

clause de retour s'écrit directement $\lambda\overrightarrow{(a' : A)}$ (x' : **I** a'$_1$ ⋯ a'$_s$) ⇒ P' où P' est P dans lequel sont réalisés les α-renommages a$_i$ -> a'$_i$ et x -> x'.

Remarque Le terme P' est une généralisation de P, les termes a$_1$ ⋯ a$_s$, x (qui se trouvent être des variables) sont des objets immuables alors que les variables a'$_1$ ⋯ a'$_s$, x' sont des abstractions pour différents termes suivant les cas. Ce sont les outils du raffinement des types dans les branches.

Par rapport au typage direct du filtrage généralisé (chapitre 3), le sens dans lequel la correspondance entre les type est faite dans les branches est en quelque sorte inversé. Alors que l'unification ou les réécritures qui la simulent plongent les types des arguments du constructeur dans les types du contexte, c'est maintenant le type de retour qui est réécrit en correspondance avec le type du constructeur. En conséquence, si le type du constructeur est trop général, le type attendu est trop général (voir section 4.4).

Si maintenant les index a$_1$ ⋯ a$_s$ de x ne sont plus nécessairement des variables, un deuxième cas favorable peut se produire. Les feuilles du squelette d'index du type de x peuvent être des variables deux à deux distinctes et de types découlant de la définition de **I**. Les constatations pour écrire des lemmes d'injection (figure 3.9) sont alors réemployables. Une clause de retour avec égalité (section 3.2.2) x = **C** a b → P a b peut s'écrire `case` x `predicate` Set `of` |λ… ⇒ Q … |λ(a' : _) (b' : _) ⇒ P a' b' … |λ… ⇒ Q `end`.

Les branches correspondant à des constructeurs dont l'index ne commence pas par **C** sont des branches impossibles. Comme expliqué section 3.4.2, le type Q peut donc a priori être choisi arbitrairement mais il est judicieux qu'il soit un type trivialement habité afin d'avoir un terme de type Q canonique. Tous les cas d'une élimination d'un inductif doivent en effet être donnés, y compris ceux qui sont impossibles pour toute exécution car de type incompatible.

Nous utiliserons dans un premier temps **True** (et **I**) (définis figure 3.4) avant de raffiner ce choix section 4.6 pour une raison d'inférence de la terminaison des points fixes, problème indépendant de celui considéré ici.

Une clause de retour du filtrage d'un terme dont le type est un motif est donc caractérisé par une formulation comme `match t as x' in` **I** p$_1$ ⋯ p$_n$ `return P with ... end` avec les p$_1$ ⋯ p$_n$ des motifs. `Coq` autorise d'ailleurs directement cette syntaxe. La structure pour donner le prédicat de retour `in` **I** p$_1$ ⋯ p$_n$ `return P` est transformée en interne en λy$_1$ ⋯ y$_n$ ⇒ « match y$_1$ ⋯ y$_n$ with
| p$_1$ ⋯ p$_n$ => P
| _ => **True** end »

Le filtrage est entre guillemets car il s'agit d'un filtrage généralisé à compiler à son tour.

Intuitivement, les termes qui composent ces clauses de retour sont un filtrage à un seul cas intéressant, celui qui isole le squelette d'index du terme filtré. Jean Francois Monin et Xiaomu Shi ont introduit le terme *diagonaliseur* pour parler d'eux dans [43].

La différence d'expressivité entre le filtrage par couverture et le filtrage par analyse de cas est assez explicite avec les clauses de retour mises sous cette forme. Le motif utilisé ici est un motif de la programmation fonctionnelle, il doit être linéaire. Le motif de l'analyse de couverture est un motif de la programmation logique qui n'a pas cette contrainte. L'identification de deux types est exactement ce qui nécessite *K*.

4.3 Diagonalisation

Pour pouvoir construire une clause de retour, il faut savoir :

- extraire le squelette d'index, mais aussi,
- abstraire les variables libres dont le type interfère avec l'objet filtré ou ses index.

Conserver les liens avec les types des variables libres introduit les coupures entrelacées. Spécialiser la clause de retour suivant le squelette d'index élimine les branches impossibles. Cela demande du filtrage dans le diagonaliseur. Il y a donc du filtrage dans la clause de retour du filtrage. Cependant le filtrage dans la clause de retour a lieu sur le type d'un index du terme filtré. Il a donc lieu sur un inductif défini strictement plus tôt dans le contexte global. La terminaison est assurée, la lisibilité du résultat par contre ne l'est pas. Nos exemples explicatifs sont incrémentaux pour ne pas noyer le lecteur.

Commençons par illustrer l'utilisation d'un squelette d'index et la reconnaissance de branches impossibles. Fournir un habitant du type **False** est et doit rester impossible. Il est pourtant possible d'écrire une preuve de ∀(H : **even** 1), **False** (pour **even** défini figure 1.12) par filtrage sur H.

La clause de retour de l'élimination d'un terme de type **even** u a la forme λ(n : **nat**) (e : **even** n) ⇒ P n e. Les constructeurs **S** et **O** sont des constantes injectives. Le diagonaliseur de la figure 4.1 permet en filtrant sur n de montrer que P 1 _ renvoie **False** comme attendu mais que P (**S** (**S** m)) (**even_SS** m) et P **O** **even_O** demandent des termes de type **True** faciles à construire.

```
Definition diag_even1 (n : nat) (e : even n) : Set :=
case n predicate λ(_ : nat) ⇒ Set of
|O:True
|S:λ(m : nat) ⇒ case m predicate λ(_ : nat) ⇒ Set of
 |O:False
 |S:λ(_ : nat) ⇒ True
 end
end.
Definition even1 (H : even (S O)) : False :=
case H
predicate λ(n : nat) (e : even n) ⇒ diag_even1 n e of
 |even_O:I
 |even_SS:λ(m : nat) ⇒ I
end.
```

Figure 4.1 – Diagonaliseur pour **even** 1

La section précédente a illustré deux techniques : généraliser le type attendu en retour de l'analyse de cas et incorporer aux clauses de retour la séparation des constructeurs. Mettons maintenant en œuvre pleinement ces constructions. Dans l'exemple précédent (x = **C** a b → P a b) les variables a b sont devenues a' b' à l'intérieur du `case`. Plus généralement, les variables qui apparaissent en feuilles du squelette d'index sont substituées autant que possible par les variables issues de l'analyse de cas. Une telle substitution n'est pas une opération sûre : si la

variable substituée apparaît dans les paramètres de l'inductif ou apparaît à plusieurs feuilles, le terme construit est mal typé. La fin de ce chapitre détaille notre heuristique (section 4.5).

Ecrire un terme de type ∀(n : **nat**) (e : **even** (**S** (**S** n))), **even** n par analyse de cas sur e met en application ce principe. Le cas **even_O** est une branche impossible car O ≠ **S**(**S** _). L'argument m de **even_SS** est utilisé lui dans le type attendu. Le prédicat de diagonalisation est

```
Definition diag_SS (n : nat) (e : even (S (S n))) : Set :=
case n predicate λ(_ : nat) ⇒ Set of
|O:True
|S:λ(x : nat) ⇒ case x predicate λ(_ : nat) ⇒ Set of
 |O:True |S:λ(m : nat) ⇒ even m end
end.
```

FIGURE 4.2 – Diagonaliseur pour inverser **evenSS**

Nous n'utilisons ici que la feuille du squelette d'index dont le type n'est pas dépendant. Dès qu'est utilisée une variable dont le type évolue au cours du raffinement du squelette d'index, une coupure entrelacée est nécessaire. En particulier, le type de la variable représentant la généralisation du terme filtré doit se raffiner au fur et à mesure que les index du terme filtré sont raffinés.

Afin d'illustrer cette affirmation, nous introduisons une nouvelle structure de données dépendante. La figure 4.3 défini la famille des ensembles finis. Le type **Fin** n représente l'ensemble $[|1 .. n|]$. Le constructeur **F1** m représente le premier élément d'un ensemble à **S** m éléments alors que **FS** m f représente le $(k+1)$-ième élément d'un ensemble à **S** m éléments si f représente le k-ième élément d'un ensemble à m éléments. Par exemple, **Fin** **O** est vide, **Fin** (**S** (**S** **O**)) contient **F1** (**S** **O**) et **FS** (**S** **O**) (**F1** **O**), etc.

```
Inductive Fin : nat → Set :=
| F1 :∀(n : nat), Fin (S n)
| FS :∀(n : nat), Fin n → Fin (S n).
```

FIGURE 4.3 – Famille inductive des ensembles de taille fixée

Avec cette définition, comment raisonner par cas sur un élément quelconque d'un ensemble non vide ? Supposons par exemple que t a le type **Fin** (**S** n) pour n fixé et que l'on souhaite raisonner par cas dessus. Le but est alors d'écrire un terme de type ∀(n : **nat**) (f : **Fin** (**S** n)), P n f. Son diagonaliseur va nécessairement débuter par les abstractions issues de la définition de la famille inductive : λ(k : **nat**) (f' : **Fin** k) ⇒ …. il faut ensuite filtrer sur k pour retrouver qu'il est non nul sans perdre le fait que f' a k éléments. La variable f' est dans le contexte, une coupure entrelacée la remet dans la conclusion. La figure 4.4 montre le résultat.

Les diagonaliseurs réalisent pour l'instant des filtrages sur **nat**, un inductif non dépendant. Les diagonaliseurs des analyses de cas du diagonaliseur étaient implicites car simples. Dans le cas général, il faut de nouveau prendre en considération dans le diagonaliseur les branches

```
Definition diag_finS (k : nat) (f' : Fin k) : Set :=
case k predicate λ(k' : nat) ⇒ Fin k' → Set of
|O:λ(fo : Fin O) ⇒ True
|S:λ(k' : nat) (fs : Fin (S k')) ⇒ P k' fs
end f'.
λ(n : nat) (f : Fin n) ⇒ case f predicate diag_finS of
|F1:λ(m : nat) ⇒ ... : P m (F1 m)
|FS:λ(m : nat) (f' : Fin m) ⇒ ... : P m (FS m f')
end
```

FIGURE 4.4 – Diagonaliseur pour ensemble non vide

impossibles, généraliser suivant les index et le terme éliminé mais aussi prévenir les désynchronisations des types des variables du contexte avec ceux des arguments du constructeur dans la branches. Bref, bâtir récursivement nos diagonaliseurs.

Considérons par exemple l'élimination d'un prédicat (H : **FLast** (**S** n) (**FS** n f)) où **FLast** exprime la propriété d'être le dernier élément d'un ensemble non élémentaire (figure 4.5) pour conclure P n f H . Il faudra construire successivement trois diagonaliseurs.

```
Inductive FLast : ∀(n : nat) (f : Fin n), Set :=
| FL1 :FLast (S O) (F1 O)
| FLS :∀(n : nat) (f : Fin n), FLast n f → FLast (S n) (FS n f).
```

FIGURE 4.5 – Dernier élément d'un ensemble

Le prédicat de diagonalisation figure 4.7 débute par un filtrage sur l'entier. L'ensemble et le prédicat ne doivent pas être introduits afin de conserver les dépendances de type. Le prédicat (diag_Fin : ∀(k : **nat**) (f : **Fin** (**S** k)) (H : **Flast** (**S** k) f), Set) doit ensuite par filtrage sur f être égal à P appliqué aux variables du dernier filtrage dans le cas **FS** et à **True** sinon. Pour conserver les correspondances de type alors qu'un constructeur apparaît dans le type de f, l'élimination a pour clause de retour le diagonaliseur diag_nat.

Ici il existe un diagonaliseur bien plus simple donné figure 4.8 qui économise l'analyse de cas sur les entiers vu qu'ils sont arguments de l'ensemble. Néanmoins, la présence de diagonaliseurs dans les diagonaliseurs est générique et absolument nécessaire en général.

Pour preuve, voici notre dernier exemple : l'inversion du prédicat d'égalité sur deux ensembles de même taille défini figure 4.6. Il met en oeuvre tous les mécanismes expliqués. Cette fois, tous sont nécessaires. Le résultat est indigeste, il est heureux que la section 4.7 donne l'algorithme formel pour le générer afin que plus jamais il ne soit écrit manuellement.

```
Inductive FinEq : ∀(n : nat), Fin n → Fin n → Set :=
| F1Eq :∀(n : nat), FinEq (S n) (F1 n) (F1 n)
| FSEq :∀(n : nat) (f1 f2 : Fin n), FinEq n f1 f2 → FinEq (S n) (FS n f1) (FS n f2).
```

FIGURE 4.6 – Prédicat d'égalité sur les éléments dans des ensembles de même taille

```
Definition diag_FLS (m : nat) : ∀(f : Fin m) (H : FLast m f), Set :=
case m predicate λ(m' : nat) ⇒
 ∀(f : Fin m') (H : FLast m' f), Set of
|O:λ(f' : Fin O) (_ : FLast O f') ⇒ True
|S:λ(k : nat) ⇒ diag_Fin k
end.
where
Definition diag_Fin (k : nat) (f : Fin (S k)) : ∀(H : FLast (S k) f), Set :=
case f predicate diag_nat of
|FS:λ(j : nat) (f0 : Fin j) ⇒
 λ(H : FLast (S j) (FS j f0)) ⇒ P j f0 H
|F1:λ(j : nat) ⇒ λ(_ : FLast (S j) (F1 j)) ⇒ True
end.
and
Definition diag_nat (i : nat) : ∀(f : Fin i), Set :=
case i predicate λ(i' : nat) ⇒ ∀(f : Fin i'), Set of
|O:λ(_ : Fin O) ⇒ True
|S:λ(j : nat) ⇒ λ(f' : Fin (S j)) ⇒
 ∀(H : FLast (S j) f'), Set
end.
```

FIGURE 4.7 – Diagonaliseur pour ∀(H : **FLast** (**S** n) (**FS** n f)), P n f H

```
λ(m : nat) (f : Fin m) ⇒
case f predicate λ(i : nat) (f' : Fin i) ⇒ FLast i f' → Set of
|F1:λ(i : nat) (H : FLast (S i) (F1 i)) ⇒ True
|FS:λ(i : nat) (f0 : Fin i) (H : FLast (S i) (FS i f0)) ⇒ P i f0
end
```

FIGURE 4.8 – Diagonaliseur optimisé pour ∀(H : **FLast** (**S** n) (**FS** n f)), P n f H

```
Definition diag_x1 (x1 : nat) : Fin x1 → Set :=
case x1 predicate λ(n2 : nat) ⇒ Fin n2 → Set of
|O:λ(_ : Fin O) ⇒ True
|S:λ(x0 : nat) ⇒ λ(f1' : Fin (S x0)) ⇒
 ∀(f2 : Fin (S x0)), FinEq (S x0) f1' f2 → Set
end.
Definition diag_x2 (x2 : nat) : Fin x2 → Set :=
case x2 predicate λ(n3 : nat) ⇒ Fin n3 → Set of
|O:λ(_ : Fin O) ⇒ True
|S:λ(x0 : nat) ⇒ λ(f2' : Fin (S x0)) ⇒ ∀(f1'' : Fin x0),
 FinEq (S x0) (FS x0 f1'') f2' → Set
end.
Definition diag_H (n0 : nat) : ∀(f1 f2 : Fin n), FinEq n f1 f2 → Set :=
case n0 predicate λ(n1 : nat) ⇒ ∀(f4 f5 : Fin n1), FinEq n1 f4 f5 → Set of
|O:λ(f4 f5 : Fin O) (_ : FinEq O f4 f5) ⇒ True
|S:λ(x : nat) ⇒ λ(f4 : Fin (S x)) ⇒
   case f4 predicate diag_x1 of
   |F1:λ(n1 : nat) ⇒ λ(f5 : Fin (S n1))
       (_ : FinEq (S n1) (F1 n1) f5) ⇒ True
   |FS:λ(n1 : nat) (f5 : Fin (S n1)) ⇒ λ(f6 : Fin (S n1)) ⇒
     (case f6 predicate diag_x2 of
     |F1:λ(n2 : nat) ⇒ λ(f7 : Fin n2)
       (_ : FinEq (S n2) (FS n2 f7) (F1 n2)) ⇒ True
     |FS:λ(n2 : nat) (f7 : Fin n2) ⇒ λ(f8 : Fin n2)
       (_ : FinEq (S n2) (FS n2 f8) (FS n2 f7)) ⇒
       FinEq n2 f8 f7
     end f5)
   end
end.
Definition invert_H (n : nat) (f1 f2 : Fin n)
 (H : FinEq (S n) (FS n f1) (FS n f2)) : FinEq n f1 f2 :=
case H predicate λ(n0 : nat) (f f0 : Fin n0) (H' : FinEq n0 f f0) ⇒ diag_n f f0 H' of
|F1Eq:λ(_ : nat) ⇒ I
|FSEq:λ(m : nat) (f' f'' : Fin m) (H0 : FinEq m f' f'') ⇒ H0
end.
```

FIGURE 4.9 – Inversion du prédicat d'égalité sur deux ensembles non élémentaires

4.4 Quand le diagonaliseur en dit trop

La construction du diagonaliseur est réalisée suivant l'intégralité du squelette d'index de l'objet filtré. Maintenant, il faut aussi se préoccuper des constructeurs de l'inductif correspondant et de leurs squelettes d'index.

Jusqu'à maintenant, ces squelettes étaient soit égaux soit incompatibles avec le squelette de l'objet filtré. En réalité, ils peuvent être plus généraux. La forme normale du type attendu dans la branche est alors une analyse de cas sur l'un des arguments du constructeur. Pour habiter le type voulu, la branche commence par l'élimination de cet argument.

Pour caricaturer, définissons un inductif ayant un prédicat comme paramètre et un entier comme index et dont l'unique constructeur n'a pour but que d'empaqueter un habitant du prédicat pour un entier donné (figure 4.10). Supposons ensuite que l'on souhaite détruire un terme de type **boite** P (**S** n) pour P et n donnés. L'élimination a la structure indiquée en figure 4.11. Il a fallu raisonner par cas sur l'argument de **contenu** afin de peupler le type demandé dans la branche puisque lui aussi est un raisonnement par cas sur cet argument.

```
Inductive boite (D : nat → Set) : nat → Set :=
| contenu :∀(n : nat), D n → boite D n.
```

FIGURE 4.10 – Une boite à D

```
Definition ouvre_boite (D : nat → Set) (n : nat) (B : boite D (S n)) : T :=
let diag_boite := λ(m : nat) ⇒
  case m predicate λ(m' : nat) ⇒ Set of
   |O:True
   |S:λ(k : nat) ⇒ T
   end
in case B predicate λ(m : nat) (_ : boite D m) ⇒ diag_boite m of
|contenu:λ(n : nat) (x : P n) ⇒
  case n predicate λ(n' : nat) ⇒ diag_boite n' of
   |O:True
   |S:λ(k : nat) ⇒ t
   end
end.
```

FIGURE 4.11 – Éliminer une grosse boite

Voilà tout le nécessaire pour construire des clauses de retours et peupler les parties impossibles des branches si les index des termes filtrés peuvent être écrits sous la forme d'une liste de motifs du filtrage à la ML.

Des situations hors de ce cadre peuvent néanmoins apparaître à tout moment, y compris dans les index des index soit dans des appels récursifs. Trouver un comportement le plus conservatif possible quand les termes sont trop généraux est indispensable pour écrire un algorithme générique utilisable en pratique.

4.5 Des clauses de retour pour éliminer tous les termes

Nous venons de voir sur des exemples les méthodes que la compilation structurelle du filtrage utilise. Au-delà de ces exemples, quels sont les filtrages pour lesquels ces méthodes s'appliquent convenablement et sont suffisantes ? La difficulté de construire une analyse de cas dépend du type de l'objet filtré.

4.5.1 Les index du type ne contiennent que des variables et des constructeurs

Pour éliminer une structure de données n'ayant pas d'index, la section 1.7 a déjà remarqué qu'aucun type n'était variable dans la clause de retour et que le problème ne posait pas de difficulté.

En présence d'index mais pas de paramètres, la question est de savoir si ces index peuvent être vus comme des motifs. Construire une clause de retour structurellement se fait en effet au moyen d'un diagonaliseur et un diagonaliseur est construit par analyse de cas sur les index. Dans un motif de la programmation fonctionnelle, les variables sont des lieurs tous distincts. En renommant toutes les occurrences de ces variables dans le type attendu en retour de l'analyse de cas, on conserve un terme correctement typé. Construire des clauses de retour ne comporte alors pas de difficulté.

En présence de paramètre et d'index simultanément, les variables apparaissant à la fois dans les index et les paramètres sont problématiques. L'occurrence dans les index prend part au diagonaliseur et va donc être α-renommé dans la clause de retour alors que celle dans les paramètres n'y prend pas part et reste constante. Comment savoir alors quelles occurrences de la variable dans le type attendu en retour de l'analyse de cas renommer et ne pas renommer pour écrire la clause de retour ?

Que se soit suite au problème d'occurrence simultané dans un paramètre d'une variable apparaissant dans un index ou de l'occurrence multiple dans les index, supposons que la condition de linéarité d'occurrence des variables ne soit pas satisfaite pour la variable o. Il faut alors réintroduire des degrés de liberté pour écrire un prédicat de retour. Nous voulons dissocier les occurrences de o dans Q (le type attendu en retour) en fonction de l'occurrence de o dans $t'_1 \cdots t'_s$ ou $w'_1 \cdots w'_s$ dont elles proviennent. Le problème se ramène concrètement à définir (Q0 : ∀… (o1 : _) … (o2 : _) …, Set) dont le type suit la signature de l'inductif et tel que Q0 … o … o … = Q.

Notre heuristique fait le choix d'utiliser autant que le typage le permet o2 puis autant que le typage le permet o1 puis o en dernier recourt. Elle est choisie pour maximiser la « typabilité » des termes générés. Quand ni o1 ni o2 ne peuvent être choisis mais qu'un o' imaginaire unifiable à la fois à o1, o2 et o le pourrait, nous avons affaire à un filtrage qui implique l'axiome K sur le type de o. o1 et o2 ne sont pas dissociables. Notre algorithme n'utilise lui que o. Il fournit ainsi un type qui ne varie pas suivant les branches et rend inexploitable l'élimination. L'information que o1 = o2 = o est perdue.

Avant d'atteindre les feuilles du squelette d'index et les problèmes de généralisation, il a fallu extraire le motif du diagonaliseur. Pour cela :

1. Nous commençons par une phase conservative :
   - Quand une feuille du squelette d'index est une variable qui n'apparaît ni dans le type attendu en retour ni dans les types des feuilles suivantes, il est remplaçable par un _ (le joker des motifs). Par exemple, pour détruire (f : **even** m) afin d'obtenir un entier, puisque m n'apparaît pas dans **nat**, il est parfaitement admissible d'écrire `case f predicate` λ(_ : **nat**) (_ : **even** _) ⇒ **nat** `of` ... `end`
   - Dans la définition d'un inductif, un index peut en déterminer un autre. Par exemple, dans **FLast**, l'entier est déterminé par l'ensemble car il en est la taille. Plus généralement, à chaque fois que le type de l'inductif est de la forme ∀... (x : ...) ... (y : **I** $t_1 \cdots t_s$), Set et que x est l'une des feuilles du squelette d'index d'un des $t_1 \cdots t_s$, y détermine x. Un index *déterminé par ailleurs* grâce à la définition de l'inductif est remplaçable par un joker si, dans le type de l'objet filtré, il s'agit d'une variable ou que l'un des index le déterminant commence par un constructeur.
   - Un index déterminé par ailleurs ne l'est pas nécessairement qu'une seule fois. Dans la définition de **FinEq** le premier index est déterminé par le second et/ou le troisième car les deux ensembles sont de même taille : l'entier donné.
     Lorsque deux index déterminant un même troisième commencent tous deux par un constructeur, le terme t correspondant à l'index déterminé apparaît comme argument des deux constructeurs. Une occurrence déterminant l'autre, t est remplaçable par un joker dans l'un des deux index déterminant en plus de remplacer l'index déterminé par _.
2. Si après cette phase, nous n'avons plus que des constructeurs, des jokers et des variables apparaissant linéairement, et hors des paramètres du type de l'objet filtré, nous voilà dans la situation optimale. Voici un motif caractérisant le problème qui va permettre la construction d'un diagonaliseur.

4.5.2 Hors du cadre pseudo-motif

Nous allons maintenant nous attarder sur des filtrages qui ne sont pas couverts par le typage primitif. Comment éliminer une donnée dont les index du type sont des termes quelconques (des sortes, des produits des analyses de cas, ...) ?

Deux altermatives existent : perdre de l'information ou réintroduire les égalités explicites que nous souhaitions éviter.

- On ramène le problème de générer une annotation de type pour une analyse de cas à un problème d'unification d'ordre supérieur

Si l'un des t' n'est pas une variable. L'algorithme tente d'abstraire t' par une variable fraîche du type de l'argument correspondant dans la définition de l'inductif. Les chances de résultat concluant sont minces et la correspondance de cet argument avec t' est perdue. Quasi systématiquement, aucune occurrence substituable n'est trouvée et le type de retour demandé est constant suivant les branches. L'information est perdue par désynchronisation entre les types comme nous l'avons expliqué lorsque nous avons étudier les conséquence de ne pas utiliser les abstractions de la clause de retour section 3.4.2.

Si ces index n'apparaissent ni dans le type d'autres index non jokerisés ni dans le type attendu en retour de l'analyse de cas, il n'y aura pas de problème de typage mais il peut néanmoins appaitre des branches impossibles non reconnues comme telles. Prenons l'exemple de (x : **nat**) et (f : **even** (plus x (**S O**))). En détruisant f pour obtenir un nouvel entier (**nat**), il est tout-à-fait possible d'écrire la clause de retour λ(_ : **nat**) ⇒ **nat**. Nous n'avons pas l'information que la branche d'**even_O** est impossible même si effectivement $x + 1 \neq 0$.

Les désagréments ont tendance à s'accumuler. Si une variable x apparaissant pourtant linéairement dans les index a un type contenant un autre index t', x ne sera pas convenablement systématiquement renommée. Dans ∀(m n : **nat**) (f1 : **Fin** m) (f2 : **Fin** n) (f3 : **Fin** (plus n m)) (H : **FLast** (plus m n) f3), eq (**Fin** (plus m n)) (append m f1 n f2) f3, il est impossible d'éliminer H car la clause de retour est mauvaise que l'on écrive λ(x : **nat**) (y : **Fin** x) ⇒ x = plus m n → **eq** (**Fin** x) (append m f1 n f2) y ou **eq** (**Fin** (plus m n)) (append m f1 n f2) y

— L'alternative est de réappliquer la stratégie des égalités explicites du chapitre précédent. Au lieu d'effacer ces termes $u_1 \cdots u_s$, on réalise une coupure entrelacée sur l'égalité en appliquant l'analyse de cas à **eq_refl** ... $y_1 \cdots$ **eq_refl** ... y_s afin de se souvenir dans les branches à quoi sont égales les feuilles du squelette d'index. Il n'y a pas moyen de tirer avantage automatiquement des ces égalités car elles n'ont pas de constructeur en tête. Lorsque le terme de preuve est construit interactivement par tactique, l'utilisateur peut réagir manuellement aux informations retournées pour construire les branches après une élimination d'une inductif. La stratégie adoptable est différente. Mieux vaut conserver toutes les informations de correspondance quitte à utiliser pour se faire des égalités (potentiellement hétérogènes) explicitement.

4.6 Digression sur les cas impossibles dans des points fixes

Il doit être fait une remarque maintenant qui ne prend du sens qu'une fois les questions du contrôle de la bonne formation des points fixes abordée (Chapitre 6)

Une analyse de cas est considérée comme un argument admissible d'un appel récursif si toutes ses branches le sont. En particulier, si un inductif n'a, comme **False**, aucun constructeur, toutes les zéros branches d'un filtrage d'un terme de ce type sont des appels récursifs admissibles. Afin que les branches impossibles soient le plus invisibles possible, il faut les peupler par des termes qui sont des appels récursifs admissibles. C'est pourquoi, plutôt que I : True, `Coq` utilise dans les cas impossibles λ(x : **False**) ⇒ `case` x `predicate` λ(_ : **False**) ⇒ **True** `of` `end` : **False** → **True**.

Ainsi, si v est un sous terme d'un argument récursif dans la définition du point fixe f, le terme f (Vhd v) est un appel récursif admissible pour la définition

```
Definition Vhd (A : Set) (n : nat) (v : vector A (S n)) : A :=
let diag := λ(n : nat) ⇒
 case n predicate λ(n' : nat) ⇒ vector A n' → Set of
  |O:λ(_ : vector O) ⇒ False → True
  |S:λ(m : nat) ⇒ λ(_ : vector (S m)) ⇒ A
```

```
  end in
case v predicate diag of
|Vnil:λ(x : False) ⇒ case x predicate λ(_ : False) ⇒ True of end
|Vcons:λ(n : nat) (h : A) (t : vector A n) ⇒ h
end.
```

mais pas pour la définition

```
Definition Vhd (A : Set) (n : nat) (v : vector A (S n)) : A :=
let diag := λ(n : nat) ⇒
 case n predicate λ(n' : nat) ⇒ vector A n' → Set of
  |O:λ(_ : vector O) ⇒ True
  |S:λ(m : nat) ⇒ λ(_ : vector (S m)) ⇒ A
  end in
case v predicate diag of |Vnil:I |Vcons:λ(n : nat) (h : A) (t : vector A n) ⇒ h
end.
```

car **I** n'est pas un sous terme !

4.7 Mécaniser la diagonalisation et compiler le filtrage

Le chapitre 2 indique comment obtenir un arbre de décision et une pile d'objet à filtrer à partir d'un filtrage. Cette partie va indiquer comment aller mécaniquement d'un arbre de décision à une succession d'analyse de cas. Toute la difficulté est de gérer les types.

Les opérations élémentaires sont toutes mutuellement récursives, c'est pourquoi l'exposé est nécessairement incrémental et non parallèle.

Les fonctions impliquées sont

- env ⊢ `decl_branche`$^{\text{numéro_branche}}_{\text{continuation_corps_branche}}$ type_branche = résultat pour construire une branche,
- env ⊢ `construire_case` terme_éliminé clause_de_retour continuation_corps_branche = résultat pour construire une analyse de cas,
- env ⊢ (type_terme_éliminé ↦ type_attendu_retour) ↘ résultat, variables_à_généraliser pour aller du type de l'objet analysé à un diagonaliseur,
- env ⊢ ‖ (terme_éliminé : index_type_terme_eliminé) ↦ [substitution]type_attendu_retour ‖ = arbre_decision, variables_à_généraliser pour construire l'arbre de décision associé au type d'un terme éliminé,
- et env ⊢ arbre_décision : type_attendu_retour ↘ résultat pour aller d'un arbre de décision à un terme du CCI.

4.7.1 Construire une analyse de cas à partir de sa clause de retour

La première opération nécessaire est la construction d'une branche dirigée par son type. Il s'agit de la fonction `decl_branche` déclarée figure 4.12. On part du type que doit avoir la branche et d'une fonction de continuation pour construire la branche. Le système regarde le type :

— S'il reconnaît une branche impossible, il peuple la branche automatiquement et n'utilise pas la continuation.
— Si le type est une analyse de cas sur l'un des arguments du constructeur, nous sommes dans la situation de la section 4.4. Le système fait une analyse de cas sur cette variable pour n'appeler la fonction de l'utilisateur que lorsque toutes les informations de typage auront été récupérées.
— Dans les autres cas, c'est à la continuation d'agir.

$$\boxed{\Gamma \vdash \mathtt{decl_branche}^{i}_{k}\ T = t}$$

$$\frac{}{\Gamma \vdash \mathtt{decl_branche}^{i}_{k}\ (\forall\ \overrightarrow{(x : S)}, \mathbf{False} \to T) = (\lambda\ \overrightarrow{(x : S)}\ (b : \mathbf{False}) \Rightarrow \mathrm{ex_falso}\ T\ b)}$$

$$\frac{\begin{array}{c}\Gamma, \overrightarrow{(z : S)} \vdash \left(x \mapsto \begin{pmatrix} \mathbf{case}\ x\ \mathbf{predicate}\ P \\ \mathbf{of}\ |b_1 \cdots |b_s\ \mathbf{end} \\ u_1\ \cdots\ u_j \end{pmatrix}\right) \searrow \mathrm{diag}, y_1 \cdots y_n \\ \Gamma, \overrightarrow{(z : S)} \vdash \mathtt{construire_case}\ x\ \mathrm{diag}\ \mathrm{k_branche} = t \end{array}}{\Gamma \vdash \mathtt{decl_branche}^{i}_{k} \begin{pmatrix} \lambda\ \overrightarrow{(z : S)} \Rightarrow \mathbf{case}\ x \\ \mathbf{predicate}\ P \\ \mathbf{of}\ |b_1 \cdots |b_s\ \mathbf{end} \\ u_1\ \cdots\ u_j \end{pmatrix} = \lambda\ \overrightarrow{(z : S)} \Rightarrow t\ y_1 \cdots y_n}$$

$$\mathrm{k_branche}(\Gamma, T, i) = t \text{ ssi } \Gamma \vdash \mathtt{decl_branche}^{i}_{k}\ T = t$$

$$\frac{}{\Gamma \vdash \mathtt{decl_branche}^{i}_{k}\ T = k\ \Gamma\ T\ i}$$

FIGURE 4.12 – Définition de la *i*ème branche dirigée par son type T puis la continuation k

La fonction `decl_branche` peuple les parties non informatives de la branche, celles qui sont issues d'un type de branche fourni par un diagonaliseur pour ne demander à l'utilisateur que les branches utiles lors de réductions. Grâce à elle, nous définissons `construire_case`, la fonction décrite figure 4.13 qui construit une analyse de cas à partir d'un diagonaliseur et d'une continuation fournissant les habitants des branches utiles.

4.7.2 D'un arbre de décision à des analyses de cas

Nous n'avons pas encore dit comment écrire une clause de retour mais nous savons répondre aux artefacs qu'elle induit dans les termes. Il est alors possible de décrire figure 4.14 comment aller d'un arbre de décision à un terme. Cette procédure a besoin du type T du

$$\boxed{\Gamma \vdash \texttt{construire_case}\ t\ diag\ k = o}$$

$$\frac{\begin{array}{c}\Gamma \vdash C_1 \in \forall\, \overrightarrow{(z_{1n} : S_{1n})}, \mathbf{I}\ u_{11} \cdots u_{1j} \\ \cdots \qquad \Gamma \vdash C_s \in \forall\, \overrightarrow{(z_{sn} : S_{sn})}, \mathbf{I}\ u_{s1} \cdots u_{sj} \\ \Gamma \vdash \texttt{decl_branche}^{i}_{k}\ (\forall\, \overrightarrow{(z_{1n} : S_{1n})}, T\ u_{11} \cdots u_{1j}\ (C_1\ z_{11} \cdots z_{1n})) = b_1 \\ \cdots \qquad \Gamma \vdash \texttt{decl_branche}^{i}_{k}\ (\forall\, \overrightarrow{(z_{sn} : S_{sn})}, T\ u_{s1} \cdots u_{sj}\ (C_s\ z_{s1} \cdots z_{sn})) = b_s\end{array}}{\Gamma \vdash \texttt{construire_case}\ t\ T\ k = \texttt{case}\ t\ \texttt{predicate}\ T\ \texttt{of}\ |b_1 \cdots |b_s\ \texttt{end}}$$

FIGURE 4.13 – Définition de l'élimination de t à partir du prédicat de retour diag suivant la continuation k

terme à construire et de l'environnement de typage Γ. Les alias présents dans les noeuds de l'arbre de décision doivent être des variables deux à deux distinctes et différentes des variables de l'environnement. Cet invariant est garanti par les fonctions qui construisent un arbre de décision au moment où elles prennent en compte les alias donnés par l'utilisateur.

$$\boxed{\Gamma \vdash st : T \searrow t}$$

$$\frac{}{\Gamma \vdash \mathrm{Tail}\ (t) : T \searrow t} \qquad \frac{\Gamma, (x : S) \vdash p : T \searrow t}{\Gamma \vdash \mathrm{Leaf}_x :: p : \forall\, (x : S), T \searrow \lambda\, (x : S) \Rightarrow t}$$

$$\frac{\begin{array}{c}\Gamma, (x : S) \vdash (x \mapsto T) \searrow diag, y_1 \cdots y_s \\ \Gamma, (x : S) \vdash \texttt{construire_case}\ x\ diag\ k_case = t\end{array}}{\Gamma \vdash \mathrm{Node}_x(st_1 \cdots st_n) : \forall\, (x : S), T \searrow \lambda\, (x : S) \Rightarrow t\ y_1 \cdots y_s}$$

$$k_case(\Gamma, T, i) = t \text{ ssi } \Gamma \vdash st_i : T \searrow t$$

FIGURE 4.14 – D'un arbre de décision à un terme

La générations des analyses de cas suit exactement la structure de l'arbre de décision :

- A la fin la substitution des alias par des variables est appliquée au membre droit de la branche
- Une feuille est une abstraction. La substitution est enrichie du nouveau nom pour l'alias.
- Si l'on sait obtenir les variables du contexte qu'il faut généraliser et le diagonaliseur qui correspondent à une élimination, nous savons construire l'analyse de cas qui correspond à un noeud.
- Un échange introduit par des abstractions les éléments de la pile jusqu'à l'élément recherché, retire des éléments introduits ceux à oublier car ils sont imposés par le typage de l'élément recherché puis réapplique l'élément recherché suivi des éléments restants. L'échange introduit comme le noeud des coupures entrelacées mais avec des termes à filtrer plus tard et non des variables libres.

4.7.3 Extraire un diagonaliseur

Il reste à construire le diagonaliseur (figure 4.16), c'est-à-dire à extraire le squelette d'index d'un terme filtré, la généralisation des variables libres dont le type dépend des feuilles de ce squelette et à tenter la meilleurs abstraction possible du type attendu en retour en fonction de ces feuilles.

$$\boxed{\Gamma \vdash \| (t : u_1 \cdots u_s) \mapsto [\sigma]c \| = p, x_1 \cdots x_s}$$

$$\frac{\Gamma \vdash \| (t : u_1 \cdots u_s, q) \mapsto [\sigma]T \| = p_i, \overrightarrow{(y_m : S_m)} \quad \forall j \neq i, p_j = \mathrm{Leaf}_{_}^{n j+m} :: \mathrm{Tail}\ (\mathbf{False} \to \mathbf{True})}{\Gamma \vdash \| (t : C_i\ u_1 \cdots u_s, q) \mapsto [\sigma]T \| = \mathrm{Node}_x(p_1 \cdots p_n), \overrightarrow{(y_m : S_m)}}$$

$$\frac{\{(a : A) \in \Gamma \mid u \text{ occurs_in } A\} = \overrightarrow{(z : S')} \quad x \text{ variable fraiche} \quad \Gamma \vdash \| (t : q, z_1 \cdots z_s) \mapsto [\sigma, u \mapsto x]T \| = p, \overrightarrow{(y : S)}}{\Gamma \vdash \| (t : u, q) \mapsto [\sigma]T \| = \mathrm{Leaf}_x :: p, \overrightarrow{(y : S)}, \overrightarrow{(z : S')}}$$

$$\frac{\{(a : A) \in \Gamma \mid t \text{ occurs_in } A\} = \overrightarrow{(z : S)} \quad x \text{ et } y_1 \cdots y_s \text{ variables fraiches} \quad \text{abstract_all_when_possible}\ (\sigma, t \mapsto x, z_1 \mapsto y_1 \cdots z_s \mapsto y_s)\ T = v}{\Gamma \vdash \| (t : \emptyset) \mapsto [\sigma]T \| = \mathrm{Leaf}_{x, y_1 \cdots y_s} :: \mathrm{Tail}\ (v), \overrightarrow{(y : S)}}$$

FIGURE 4.15 – Construction du diagonaliseur

La méthode pour trouver la meilleure abstraction d'un terme suivant des termes donnés est due à Chung-Kil Hur. Le problème est le suivant : Pour $u_1 \cdots u_m$ une liste de termes de types respectifs $S_1 \cdots S_m$ et un terme t de type T, trouver t' tel que $t'\ u_1 \cdots u_m = t$. Le terme t' doit être bien typé. [1] Les termes $u_1 \cdots u_m$ doivent en outre apparaître le moins possible dans t'. Nous ne chercherons pas à qualifier formellement « le moins possible ». Celui-ci dissimule en effet la caractérisation de l'expressivité du filtrage sans l'axiome *K*, problème fondamental que ce manuscrit ne résout pas.

A l'origine de l'algorithme réside un judicieux détournement. Coq offre à l'utilisateur un mécanisme d'arguments implicites. Si une constante attend par exemple comme argument un entier *n* puis un **Fin** *n*, quand l'utilisateur donne l'ensemble, il donne sa taille. Le premier argument étant nécessairement cette taille, l'utilisateur n'a pas besoin de la donner explicitement, le système peut l'inférer. Le premier argument est définissable comme implicite donc non donné par l'utilisateur et le système remplace ce "trou" par le terme adéquat.

Si u de type S apparaît dans t, remplaçons toutes les occurrences de u dans t par des "trous" qui se nomment en réalité des *variables existentielles*. Demandons au système de résoudre ses variables existentielles. Là où u est indispensable par typage, le système va le remettre.

1. D'ailleurs, obtenir son type demande déjà du travail.

Ailleurs, remplaçons les variables restantes par une variable y fraîche pour obtenir t'. Le terme λ(y : S) ⇒ t' est bien typé et convertible à t quand il est appliqué à u.

Nous avons obtenu une solution d'abstraction sur un terme. Pour une liste de termes la solution est similaire mais compliquée par les dépendances de type entres les termes à abstraire. Il faut en effet commencer par le dernier élément du futur télescope afin qu'ensuite son type soit lui aussi généralisé selon les termes dont il dépend.

La fonction `abstract_all_when_possible` doit donner une réponse bien typée et telles que si $\sigma \circ \sigma' = id$ (σ' est une substitution standard des variables vers les termes) alors σ'(`abstract_all_when_possible` σ T) = T.

$$\boxed{\Gamma \vdash (t \mapsto c) \searrow out, x_1 \cdots x_s}$$

$$\frac{\begin{array}{c}\Gamma \vdash t \in \mathbf{I}\ a_1 \cdots a_m\ u_1 \cdots u_s \\ \Gamma \vdash \mathbf{I}\ a_1 \cdots a_m \in \forall\ \overrightarrow{(x : S)}, \mathrm{Set} \qquad \Gamma \vdash \|\ (t : u_1 \cdots u_s) \mapsto [\varepsilon]c\ \| = st, \overrightarrow{(y_n : T_n)} \\ \Gamma \vdash st : \forall\ \overrightarrow{(x : S)}\ (z : \mathbf{I}\ a_1 \cdots a_m\ x_1 \cdots x_s)\ \overrightarrow{(y_n : T_n)}, \mathrm{Set} \searrow o\end{array}}{\Gamma \vdash (t \mapsto c) \searrow o, y_1 \cdots y_n}$$

FIGURE 4.16 – Prédicat de retour out de l'élimination de t pour obtenir c

4.8 Eléments de correction

Notre algorithme génère-t-il des termes bien typés ? Nous ne pouvons pas réellement exprimer cette propriété. Nous ne sommes pas capable de donner une règle de typage primitive pour le filtrage généralisé n'impliquant pas l'axiome *K*. La recherche sur ce sujet est très active au moment de l'écriture de ce manuscrit mais elle n'a pas encore abouti à une caractérisation. Nous ne pouvons donc pas énoncer de propriété telle que « si le filtrage est bien typé, le terme du CCI généré par l'algorithme ci-dessus est bien typé ». Or, si l'utilisateur écrit un filtrage qui n'a pas de sens, le résultat de sa compilation sera mal-typé. Par ailleurs, notre algorithme finissant par une phase heuristique, il y a peu d'espoir de pouvoir obtenir un résultat général. Cette question n'est néanmoins pas critique. DeBruijn a introduit un principe essentiel en informatique théorique : plutôt que certifier la génération de témoins de manière complexe, assurons de manière simple a posteriori la correction du témoin exhibé. Nous adoptons exactement cette démarche car le noyau de `Coq` (un vérificateur de type du CCI) vérifie le bon typage de chacun des termes que nous générons. Il n'y a pas besoin de faire confiance à notre algorithme du point de vu du typage.

Le contrôle du terme généré par le vérificateur de type n'exonère pas de toute vérification. L'utilisateur souhaite que le comportement calculatoire de la chaîne d'analyse de cas soit celui du filtrage généralisé dont elle découle. Cette *adéquation* entre le terme généré et son origine est correcte si la règle de réduction donné section 2.1 est simulable sur le terme compilé au moyen de $\beta\iota$-réductions du CCI.

Cette preuve est réalisée en deux étapes :

- La simulation du filtrage par des réductions d'arbres de décision (figure 2.7). Cette preuve est identique au cas de ML traité par [38].
- La simulation des réduction d'arbres de décisions par les analyses de cas construites. Le lemme cruciale est le suivant : lorsqu'une analyse de cas supplémentaire sur l'un des arguments a été générée dans une branche pour raffiner son type (la situation expliquée section 4.4), cette analyse de cas va être systématiquement réduite lorsqu'elle arrive en tête. Ce lemme se prouve en raisonnant sur la forme de l'argument : l'argument a nécessairement une forme qui va réduire l'analyse de cas pour que le terme éliminé ait le bon type. Pour les autres situations, nos manipulations n'ont concerné que les annotations de types et la preuve est immédiate.

Jean-François Monin avait montré l'apport d'une telle manière d'écrire des termes en `Coq`. Nous apportons la possibilité de les écrire automatiquement. En ce sens, cette partie constitue un pas en avant sur le chemin de la programmation certifiée.

5 Simplifier un terme

Écrire sans assistance un terme complexe dans un langage à types riches est une chose ardue. Des systèmes pour guider l'écriture de programmes ont donc été réalisés. Que ce soit au moyen de tactiques [21] ou d'un mode d'édition interactif [41], une fonctionnalité clé est de pouvoir écrire des termes « à trous » puis de demander au système quel doit être le type des termes à mettre dans ces trous.

L'utilisateur n'a pas à fournir un terme de type syntaxiquement égal à celui attendu mais un type qui lui est convertible. Pour faciliter la compréhension, il est même souhaitable que le système n'affiche pas le type qu'il a inféré pour le trou mais plutôt une forme réduite.

Ce chapitre propose un algorithme pour obtenir une forme réduite d'un terme. Nous parlons de forme réduite et non de forme normale car la forme normale n'est pas le représentant le plus lisible d'une classe d'équivalence pour la conversion.

En effet, pour un humain, une constante (au nom bien choisie) est plus explicite que sa définition. Une δ-expansion ne doit avoir lieu que si elle induit d'autres réductions.

```
Definition plus : nat → nat → nat :=
 fix₂ (pl : nat → nat → nat := λ(m n : nat) ⇒
  case n predicate λ(_ : nat) ⇒ nat of
  | O : m | S : λ(n' : nat) ⇒ S (pl m n')
  end).
Definition succ : nat → nat := plus (S O).
```

FIGURE 5.1 – L'addition et le successeur

De plus, dans le CCI tel qu'il est défini actuellement, lors de normalisations impliquant des points fixes, déplier puis replier une constante est nécessaire. Considérons par exemple la définition de plus donnée figure 5.1. La forme normale de plus (**S** (**S** x)) y n'est pas **S** (**S** (plus x y)) mais :

> **S** (**S** (fix$_1$ (pl : **nat** → **nat** → **nat** := λ(m n : **nat**) ⇒ `case` m `predicate` λ(_ : **nat**) ⇒ **nat** `of` | **O** : n | **S** : λ(m' : **nat**) ⇒ S (pl m' n) `end`) x y))

Après une présentation de la méthode de normalisation utilisée, nous détaillerons un moyen d'obtenir la forme escomptée.

5.1 Normalisation en appel par nom

Suivant les schéma de compilation usuels (voir par exemple [26, 36]), la réduction d'un terme est réalisée en le traduisant vers un langage particulier (l'évaluation), en calculant

dans ce langage (le calcul) puis en retraduisant le résultat vers un terme (la réification). Notre langage cible est un *état de machine abstraite*. Un état de machine abstraite est la paire d'un terme et d'une *pile* où sont stockés les destructeurs de type à appliquer au terme.

La syntaxe des piles est donnée par

$$\mathrm{sk} ::= \emptyset \mid \circ\, t, \mathrm{sk} \mid \mathrm{Zcase}_{T}(u_1 \cdots u_s), \mathrm{sk} \mid \mathrm{Zfix}\ (i, (f : T := t) :: u_1 \cdots u_s), \mathrm{sk}$$

$\circ\, t$ représente un argument attendant d'obtenir la fonction à laquelle il est appliqué.

$\mathrm{Zcase}_{T}(u_1 \cdots u_s)$ stocke les branches d'une analyse de cas en attendant de connaître le constructeur par lequel débute le terme filtré.

$\mathrm{Zfix}\ (i, (f : T := t) :: u_1 \cdots u_s)$ est un point fixe attendant de savoir si son argument récursif s'évalue sur un constructeur appliqué pour se déplier.

L'évaluation débute en plaçant le terme à réduire en face de la pile vide ($\emptyset$). Les destructeurs de termes sont empilés. Lorsqu'un constructeur de type est atteint, la tête de la pile est inspectée pour déclencher si possible un calcul.

Les règles de calcul pour la β-réduction de $\lambda(x : T) \Rightarrow t$ sont :

$$\begin{array}{rl} \circ\, u, \mathrm{sk}\ \ -[t] \xrightarrow{(x:T)}_{k} & k\ t[u/x]\ \mathrm{sk} \\ \emptyset\ \ -[t] \xrightarrow{(x:T)}_{k} & \langle \lambda(x : T) \Rightarrow t \mid \emptyset \rangle \end{array}$$

La continuation k du calcul sera la fonction d'évaluation.

Pour les règles ι et ϕ, il faut extraire de la tête de pile les arguments du constructeur (les nœuds $\circ\, t$). Nous notons $t_1 \cdots t_n \oplus \mathrm{sk}$ une pile constituée des nœuds applications $\circ\, t_1 \cdots \circ\, t_n$ puis de sk (ne débutant pas par une application). Les règles de calcul pour le i[e] constructeur $\mathbf{C}_i$ sont

$$\begin{array}{rl} t_1 \cdots t_s \oplus \mathrm{Zcase}_{T}(u_1 \cdots u_n), \mathrm{sk} \xrightarrow{i}_{k} & k\ u_i\ (\circ\, t_1 \cdots \circ\, t_s, \mathrm{sk}) \\ t_1 \cdots t_s \oplus & \\ \mathrm{Zfix}\ (j, (f : T := v) :: u_1 \cdots u_{j-1}), \mathrm{sk} \xrightarrow{i}_{k} & k\ v[\mathrm{fix}_j\ (f : T := v)/f] \\ & (\circ\, u_1 \cdots \circ\, u_{j-1}, \circ\, (\mathbf{C}_i\ t_1 \cdots t_s), \mathrm{sk}) \\ t_1 \cdots t_s \oplus \emptyset \xrightarrow{i}_{k} & \langle \mathbf{C}_i \mid \circ\, t_1 \cdots \circ\, t_s, \emptyset \rangle \end{array}$$

Le calcul prend encore une fois sa continuation k en argument afin que tous les appels de fonction dans la machine abstraite soient récursifs terminaux.

L'évaluation suit les règles de réécriture de la figure 5.2. La continuation k attendue par les opérations de calculs est la vision fonctionnelle des règles d'évaluations. Formellement, $k\ t\ \mathrm{sk} = \langle v \mid \mathrm{sk}' \rangle$ si et seulement si $\Delta \vdash \langle t \mid \mathrm{sk} \rangle \downarrow_{\mathrm{CBN}} \langle v \mid \mathrm{sk}' \rangle$.

Une fois qu'il n'y a plus de coupure en tête du terme, l'état de machine abstraite est réifié pour retrouver un terme en replaçant autour de la valeur obtenue les destructeurs de termes restant. Le processus est décrit figure 5.3

Ce procédé permet d'obtenir la forme normale de tête faible d'un terme par un calcul en appel par nom. Pour obtenir la forme normale forte, il faut traverser les fonctions et les

$$\boxed{\Delta \vdash \langle tm_i \mid sk_i \rangle \downarrow_{CBN} \langle tm_o \mid sk_o \rangle}$$

$$\frac{\Delta \vdash \langle t \mid \circ\, u, sk \rangle \downarrow_{CBN} \langle v \mid sk' \rangle}{\Delta \vdash \langle t\ u \mid sk \rangle \downarrow_{CBN} \langle v \mid sk' \rangle} \qquad \frac{sk -[t] \rightarrow_k^{(x : T)} \langle v \mid sk' \rangle}{\Delta \vdash \langle \lambda(x : T) \Rightarrow t \mid sk \rangle \downarrow_{CBN} \langle v \mid sk' \rangle}$$

$$\frac{\Delta \vdash \langle t \mid \mathrm{Zcase}_P(u_1 \cdots u_s), sk \rangle \downarrow_{CBN} \langle v \mid sk' \rangle}{\Delta \vdash \langle \texttt{case}\ t\ \texttt{predicate}\ P\ \texttt{of}\ u_1 \cdots u_s\ \texttt{end} \mid sk \rangle \downarrow_{CBN} \langle v \mid sk' \rangle}$$

$$\frac{\Delta \vdash \langle u_i \mid \mathrm{Zfix}\ (i, (f : T := t) :: u_1 \cdots u_{i\text{-}1}), sk \rangle \downarrow_{CBN} \langle v \mid sk' \rangle}{\Delta \vdash \langle \mathsf{fix}_i\ (f : T := t) \mid \circ\, u_1 \cdots \circ\, u_i, sk \rangle \downarrow_{CBN} \langle v \mid sk' \rangle}$$

$$\frac{}{\Delta \vdash \langle \mathsf{fix}_i\ (f : T := t) \mid u_1 \cdots u_{i\text{-}k} \oplus \emptyset \rangle \downarrow_{CBN} \langle \mathsf{fix}_i\ (f : T := t) \mid u_1 \cdots u_{i\text{-}k} \oplus \emptyset \rangle}\ k \leq i$$

$$\frac{}{\Delta \vdash \langle x \mid sk \rangle \downarrow_{CBN} \langle x \mid sk \rangle} \qquad \frac{}{\Delta \vdash \langle \mathbf{I} \mid \circ\, t_1 \cdots \circ\, t_s, \emptyset \rangle \downarrow_{CBN} \langle \mathbf{I} \mid \circ\, t_1 \cdots \circ\, t_s, \emptyset \rangle}$$

$$\frac{\Delta, (c : T := t), \Delta' \vdash \langle t \mid sk \rangle \downarrow_{CBN} \langle v \mid sk' \rangle}{\Delta, (c : T := t), \Delta' \vdash \langle c \mid sk \rangle \downarrow_{CBN} \langle v \mid sk' \rangle} \qquad \frac{}{\Delta \vdash \langle S \mid \emptyset \rangle \downarrow_{CBN} \langle S \mid \emptyset \rangle}$$

$$\frac{}{\Delta \vdash \langle \forall(x : S), T \mid \emptyset \rangle \downarrow_{CBN} \langle \forall(x : S), T \mid \emptyset \rangle} \qquad \frac{sk \xrightarrow{i}_k \langle v \mid sk' \rangle}{\Delta \vdash \langle C_i \mid sk \rangle \downarrow_{CBN} \langle v \mid sk' \rangle}$$

FIGURE 5.2 – Call By Name abstract machine

$$\boxed{\langle u \mid sk \rangle \uparrow v}$$

$$\frac{}{\langle t \mid \emptyset \rangle \uparrow t} \qquad \frac{\langle \texttt{case}\ t\ \texttt{predicate}\ T\ \texttt{of}\ u_1 \cdots u_s\ \texttt{end} \mid sk \rangle \uparrow v}{\langle t \mid \mathrm{Zcase}_T(u_1 \cdots u_s), sk \rangle \uparrow v} \qquad \frac{\langle t\ u \mid sk \rangle \uparrow v}{\langle t \mid \circ\, u, sk \rangle \uparrow v}$$

$$\frac{\langle \mathsf{fix}_i\ (f : T := u)\ t_1 \cdots t_{i\text{-}1}\ t \mid sk \rangle \uparrow v}{\langle t \mid \mathrm{Zfix}\ (i, (f : T := u) :: t_1 \cdots t_{i\text{-}1}), sk \rangle \uparrow v}$$

FIGURE 5.3 – Réinterprétation d'un état comme une valeur

produits afin de normaliser leur corps puis appliquer récursivement le même processus aux termes qui apparaissent dans les éléments de la pile avant de réaliser la réification.

Notre machine est inefficace car elle réalise les substitutions une par une au lieu d'utiliser un environnement. Cette machine avec des environnements est une machine de Krivine [33] ou plutôt une machine de Krivine avec types algébriques [6] puisqu'elle manipule des inductifs. Nous la présenterons section 5.3. Avant cela, traitons le cœur de notre contribution, les repliages de constante.

5.2 Trace des constantes dépliées

Nous ajoutons aux états de machine de la section précédente une liste des constantes dépliées convertibles au terme de l'état. Cette liste va permettre de savoir quelle constante replier.

Nous utilisons une liste car nous souhaitons traiter ce que nous nommons une *cascade* de constantes. Il s'agit d'une situation où une constante est définie grâce à une autre constante. Par exemple, nombre de fonctions (telles que incr_ids figure 5.1) sur les listes (celle du langage définie figure 1.11) sont définies grâce aux itérateurs sur les listes (comme list_map) qui eux sont définis par point fixe et analyse de cas .

```
Definition list_map (A B : Set) (f : A → B) : list A → list B :=
fix₁ (map : list A → list B := λ(l : list A) ⇒ case l
predicate λ(_ : list A) ⇒ list B of
|nil:nil B
|cons:λ(h : A) (t : list A) ⇒ cons (f h) (map t)
end).
Definition incr_ids : list nat → list nat :=
list_map nat nat succ.
```

Figure 5.4 – Exemple d'itérateur et d'une de ses instances

Formellement, un *dépliage* est un triplet constitué d'un nom de constante et de deux listes de termes appelées les *paramètres* et les *arguments*.

Une liste de dépliage p est un *repliage* du terme t noté t ⊩ p si pour tous ses éléments la constante appliquée aux paramètres est convertible à t appliqué aux arguments.

Le point fixe pl a pour dépliage (plus,[],[]) qui n'a ni argument ni paramètre mais on comprend le sens des paramètres pour donner un dépliage du point fixe map : (list_map,[S, T, t],[]) (avec S, T et t les valeurs de A, B et f dans le cas considéré). L'intérêt des arguments est illustré par : pl a pour dépliage alternatif (succ,[],[**S O**])

Les primitives dont nous avons besoin sur les listes de dépliages sont définies telles que

- Si t u ⊩ p alors t ⊩ `add_arg` u p.
- Pour u donné, si λ(x : T) ⇒ t ⊩ p et pour tout $(c, \mathrm{params}, \mathrm{args}) \in p$, on a soit (args = u, _) soit (args = ∅), alors t[u/x] ⊩ `add_param` u p.
- Si c ⊩ p et $\Delta(c) =$ t alors t ⊩ `add_cst` c p.

par

```
(** val add_arg : term → cst_stk → cst_stk *)
let add_arg arg = List.map (fun (a,b,c) → (a,b,c @ [arg]))

(** val add_param : term → cst_stk → cst_stk *)
let add_param param p = List.map (fun (a,b,c) →
  match c with |[] → (a,b @ [param],c) |_ : :q → (a,b,q)) p

(** val add_cst : cst → cst_stk → cst_stk *)
let add_cst cst p = (c,[],[]) : : p
```

Les noeuds de la pile qui détruisent un inductif sont modifiés. Une liste de dépliage est ajoutée. Elle représente un repliage de la réification du terme et de la pile jusqu'à ce noeud inclus. Il existe une fonction *effacement* triviale qui permet de retrouver une pile de machine appel par nom à partir d'une pile avec les traces en supprimant ces annotations.

$$sk ::= \emptyset \mid \circ\, t, sk \mid Zcase_T(u_1 \cdots u_s/p), sk \mid Zfix\ (i, (f : T := t)/p :: u_1 \cdots u_s), sk$$

Les étapes de calcul et d'évaluation (figure 5.5) sont modifiées de telle sorte à maintenir un repliage du terme de l'état de la machine. Nous nommons RN cette nouvelle machine. L'évaluation enregistre aussi les repliages des destructeurs d'inductif lorsqu'elle les empile.

$$\frac{\circ\, u, sk}{p} \;-[t] \rightarrow^{(x:T)}_{k}\; k\ (\text{add_param}\ u\ p)\ (t[u/x])\ sk$$

$$\frac{\emptyset}{p} \;-[t] \rightarrow^{(x:T)}_{k}\; \langle \lambda(x : T) \Rightarrow t \mid \emptyset \rangle$$

Maintenant que la machinerie est en place, le calcul sur les inductifs et la réification peuvent en tirer avantage. Pour cela, nous définissons la notion de *meilleur repliage*.

Lors du calcul de succ (**S** m), le repliage du point fixe à l'intérieur de sa définition est [(plus,[],[]) ;(succ,[],[**S O**])]. La constante dépliée en premier, le premier élément de la cascade, est en fond de repliage. Nous voulons l'utiliser autant que possible pour remplacer le terme déplié. C'est possible si le terme à replier est appliqué aux arguments du dépliage (c'est bon pour succ si le premier argument de pl est **S O**). Si ce n'est pas le cas, nous essayons avec le dépliage suivant (plus dans notre exemple). Les fonctions qui choisissent le meilleur repliage sont refold_term et refold_state. La fonction refold_term travaille comme une substitution qui regarde les arguments de la variable à substituer pour choisir par quoi substituer. La fonction refold_state travaille elle sur un état de machine abstraite. Elle inspecte les termes en tête de pile pour choisir par quoi elle va remplacer le terme.

Par construction, le nombre d'arguments des dépliages dans un repliage est une fonction croissante. Un dépliage n'ayant pas d'argument peut toujours être utilisé car il n'y a alors aucune condition sur le contexte. Nous ne conservons donc en pratique qu'un seul dépliage sans argument en tête du dépliage. Lors du calcul de incr_ids (**cons** x y), le repliage du point fixe est [(map_list, [**nat** ;**nat** ;succ], []) ;(incr_ids,[],[])] mais nous ne gardons que [(incr_ids,[],[])] qui est toujours utilisable.

Lors du dépliage d'un point fixe, le meilleur repliage du point fixe est utilisé dans la substitution plutôt que la définition du point fixe.

$$\boxed{\Delta \vdash_p \langle tm_i \mid sk_i\rangle \downarrow_{RN} \langle tm_o \mid sk_o\rangle}$$

$$\frac{\overset{s}{p} \; -[t] \rightarrow_{k}^{(x : T)} \; \langle v \mid s'\rangle}{\Delta \vdash_p \langle \lambda(x : T) \Rightarrow t \mid s\rangle \downarrow_{RN} \langle v \mid s'\rangle} \qquad \frac{\Delta \vdash_{\mathrm{add_arg}\ u\ p} \langle t \mid \circ\, u, s\rangle \downarrow_{RN} \langle v \mid s'\rangle}{\Delta \vdash_p \langle t\ u \mid s\rangle \downarrow_{RN} \langle v \mid s'\rangle}$$

$$\frac{}{\Delta \vdash_p \langle \forall(x : S), T \mid \emptyset\rangle \downarrow_{RN} \langle \forall(x : S), T \mid \emptyset\rangle} \qquad \frac{sk \overset{i}{\rightarrow}_k \langle v \mid sk'\rangle}{\Delta \vdash_p \langle C_i \mid sk\rangle \downarrow_{RN} \langle v \mid sk'\rangle}$$

$$\frac{\Delta \vdash_{\emptyset} \langle t \mid \mathrm{Zcase}_T(u_1 \cdots u_s/p), sk\rangle \downarrow_{RN} \langle v \mid sk'\rangle}{\Delta \vdash_p \langle \texttt{case}\ t\ \texttt{predicate}\ T\ \texttt{of}\ u_1 \cdots u_s\ \texttt{end} \mid sk\rangle \downarrow_{RN} \langle v \mid sk'\rangle}$$

$$\frac{\Delta \vdash_{\emptyset} \langle u_i \mid \mathrm{Zfix}\ (i, (f : T := t)/p :: u_1 \cdots u_{i-1}), sk\rangle \downarrow_{RN} \langle v \mid sk'\rangle}{\Delta \vdash_p \langle \mathsf{fix}_i\ (f : T := t) \mid \circ\, u_1 \cdots \circ\, u_i, sk\rangle \downarrow_{RN} \langle v \mid sk'\rangle}$$

$$\frac{}{\Delta \vdash_p \langle \mathsf{fix}_i\ (f : T := t) \mid u_1 \cdots u_{i-k} \oplus \emptyset\rangle \downarrow_{RN} \langle \mathsf{fix}_i\ (f : T := t) \mid u_1 \cdots u_{i-k} \oplus \emptyset\rangle}$$

$$\frac{}{\Delta \vdash_p \langle x \mid sk\rangle \downarrow_{RN} \langle x \mid sk\rangle} \qquad \frac{}{\Delta \vdash_p \langle \mathbf{I} \mid \circ\, t_1 \cdots \circ\, t_s, \emptyset\rangle \downarrow_{RN} \langle \mathbf{I} \mid \circ\, t_1 \cdots \circ\, t_s, \emptyset\rangle}$$

$$\frac{\Delta, (c : T := t), \Delta' \vdash_{\mathrm{add_cst}\ c\ p} \langle t \mid sk\rangle \downarrow_{RN} \langle v \mid sk'\rangle}{\Delta, (c : T := t), \Delta' \vdash_p \langle c \mid sk\rangle \downarrow_{RN} \langle v \mid sk'\rangle} \qquad \frac{}{\Delta \vdash_p \langle s \mid \emptyset\rangle \downarrow_{RN} \langle s \mid \emptyset\rangle}$$

FIGURE 5.5 – Refoldant Call By Name abstract machine

$$\begin{array}{rcl}
t_1 \cdots t_s \oplus \mathrm{Zcase}_T(u_1 \cdots u_s/p), sk & \xrightarrow{i}_k & k\ \emptyset\ u_i\ (\circ\ t_1 \cdots \circ\ t_s, sk) \\
t_1 \cdots t_s \oplus & \xrightarrow{i}_k & k\ \emptyset\ v[\mathrm{refold_term}\ p/f] \\
\mathrm{Zfix}\ (j, (f : T := v)/p :: u_1 \cdots u_{j-1}), sk & & (\circ\ u_1 \cdots \circ\ u_{j-1}, \circ\ (C_i\ t_1 \cdots t_s), sk) \\
t_1 \cdots t_s \oplus \emptyset & \xrightarrow{i}_k & \langle C_i \mid \circ\ t_1 \cdots \circ\ t_s, \emptyset \rangle
\end{array}$$

Lorsqu'un éliminateur d'inductif est replié, le meilleur repliage est utilisé plutôt que la construction elle-même (figure 5.6).

$$\boxed{\langle u \mid sk \rangle \uparrow v}$$

$$\frac{}{\langle t \mid \emptyset \rangle \uparrow t} \qquad \frac{\langle \mathrm{refold_state}\ p\ (\mathtt{case}\ t\ \mathtt{predicate}\ T\ \mathtt{of}\ u_1 \cdots u_s\ \mathtt{end}) \mid sk \rangle \uparrow v}{\langle t \mid \mathrm{Zcase}_T(u_1 \cdots u_s/p), sk \rangle \uparrow v}$$

$$\frac{\langle t\ u \mid sk \rangle \uparrow v}{\langle t \mid \circ\ u, sk \rangle \uparrow v} \qquad \frac{\langle \mathrm{refold_state}\ p\ (\mathsf{fix}_i\ (f : T := u)\ t_1 \cdots t_{i-1}\ t) \mid sk \rangle \uparrow v}{\langle t \mid \mathrm{Zfix}\ (i, (f : T := u)/p :: t_1 \cdots t_{i-1}), sk \rangle \uparrow v}$$

FIGURE 5.6 – Réification du repliage d'une valeur

La correction de notre machine est garantie par le théorème : Pour tout terme t et toute pile de machine sk, si $\Delta \vdash_p \langle t \mid sk \rangle \downarrow_{RN} \langle v \mid s \rangle$, $t \Vdash p$ et $\Delta \vdash \langle t \mid \text{effacement}(sk) \rangle \downarrow_{CBN} \langle v' \mid s' \rangle$ alors $\Delta \vdash v \mapsto_{\beta\delta} v'$ et les termes apparaissant dans l'effacement de s se réduisent modulo $\beta\delta$ vers le terme à la même place dans s'. Toutes les réductions ι et ϕ possibles ont été réalisées.

5.3 Refolding Algebraic Krivine Abstract Machine

La machine présentée jusqu'ici fonctionne mais elle est inefficace. Elle est calculatoirement inefficace comme nous l'avons déjà remarqué section 5.1. Pire, sa capacité à réaliser les repliages est limitée. Ces deux sous-optimalitées se corrigent en améliorant les substitutions.

Pour comprendre le problème avec les repliages, revenons sur l'exemple succ (**S** m).

Le système répond **S** (plus (**S** **O**) n) et non **S** (succ n) car le dépliage du point fixe a lieu avant la substitution de m par **S** **O** dans le corps du point fixe. L'argument de l'appel récursif au moment de la substitution ne correspond pas à l'argument demandé pour replier par succ, ce choix est donc éliminé.

D'autre part, en repliant par la meilleure constante à chaque fois, les δ-réductions ont lieu à chaque fois. Lors de la réduction de plus m (**S** (**S** n)), la constante plus est dépliée deux fois. En retardant la substitution, les alternatives peuvent toutes deux être conservées afin que dans un calcul, le point fixe soit utilisé et que durant la réification, la constante donne un terme concis.

Nous modifions notre machine pour qu'elle manipule non plus des termes mais des clôtures, c'est-à- dire des paires (terme, substitution).

La grammaire des substitutions est constituée de trois types de noeuds, le premier pour les termes non évalués, le deuxième pour les constructeurs déjà calculés (car ils sont l'argument

récursif d'un point fixe) et le troisième pour les points fixes dépliés qui stockent la cascade de constantes dont ils sont issus.

$$\sigma ::= \emptyset \mid \{x := \text{Sapp } \sigma'(t), \sigma\} \mid \left\{x := \text{Scstr}_j\ \sigma_1(t_1) \cdots \sigma_s(t_s), \sigma\right\} \mid \{x := \text{Sfix}_i\ \sigma'(f : T := t)/\text{cst}, \sigma\}$$

La pile est modifiée car tous les termes deviennent des clôtures. Comme dans les substitutions, un noeud particulier est ajouté pour des valeurs déjà calculées. Une fois réduit sur un constructeur, l'argument récursif d'un point fixe doit en effet être réempilé afin de procéder au dépliage du point fixe. La fonction l⊕sk doit maintenant aussi extraire ces noeuds spéciaux.

Remarque Avec des valeurs qui sont des clôtures et non plus des termes, un comportement remarquable de la machine apparaît. Le calcul de l'argument récursif pour autoriser le dépliage d'un point fixe introduit de l'appel par valeur au sein de notre machine en appel par nom. Avant d'avoir des substitutions, nous réifions implicitement cette valeur avant de la remettre dans la pile.

$$\text{sk} ::= \emptyset \mid \circ\ \sigma(t), \text{sk} \mid \text{Zcase}_T(\sigma(u_1 \cdots u_s)/\text{cst}), \text{sk} \mid \text{Zfix}\ (i, \sigma(f : T := t)/\text{cst} :: \sigma_1(u_1) \cdots \sigma_{i\text{-}1}(u_{i\text{-}1})), \text{sk} \mid \text{Zcstr}_i\ \sigma_1(t_1) \cdots \sigma_s(t_s), \text{sk}$$

Les repliages ne sont pas profondément différents hormis qu'ils manipulent des clôtures.

```
(** val add_arg : term → subst → cst_stk → cst_stk *)
let add_arg arg sigma p = List.map (fun (a,b,c) → (a,b,c @ [arg, sigma])) p
(** val add_param : term → subst → cst_stk → cst_stk *)
let add_param param sigma p = List.map (fun (a,b,c) →
  match c with |[] → (a,b @ [param, sigma],c) |_ ::q → (a,b,q)) p
(** val add_cst : cst → cst_stk → cst_stk *)
let add_cst cst p = (c,[],[]) :: p
```

Le calcul ne fait plus de substitution, il enrichit la substitution de la clôture.

Les règles de réécriture sont moins claires du fait des clôtures mais seul change réellement le fait d'interroger la substitution quand une variable est atteinte.

A la fin de l'évaluation, il faut retourner d'une clôture à un terme en appliquant effectivement la substitution avant de réifier l'état de la machine. La propagation d'une substitution σ dans un terme t pour obtenir u est notée $u \ltimes \sigma$.

Cette nouvelle machine est meilleure que la précédente si elle réalise plus de repliage tout en calculant toujours toutes les ϕ expansions et les ι réductions possibles. Le théorème le garantissant serait qu'en appliquant les substitutions à la sortie de la machine, on ait un terme qui se $\beta\delta$ réduise sur la sortie de la machine appel par nom avec repliage.

5.4 Configurabilité

Trouver la manière optimale de réduire une constante n'est pas du ressort d'un ordinateur car la notion elle-même n'est pas définie. Certaines constantes masquent des preuves sans

$$\begin{array}{r}\circ\ \sigma(u),\ sk\\ p\end{array} \ -[t] \overset{\sigma'(x\,:\,T)}{\longrightarrow}_k \ \begin{array}{l}k\ (\mathrm{add_param}\ u\ \sigma\ p)\ t\\ (\{x := \mathrm{Sapp}\ \sigma(u),\ \sigma'\})\ sk\end{array}$$

$$\begin{array}{r}\mathrm{Zcstr}_i\ \sigma_1(t_1)\cdots\sigma_s(t_s),\ sk\\ \emptyset\end{array} \ -[t] \overset{\sigma\ (x\,:\,T)}{\longrightarrow}_k \ \begin{array}{l}k\ \emptyset\ t\\ (\{x := \mathrm{Scstr}_i\ \sigma_1(t_1)\cdots\sigma_s(t_s),\ \sigma\})\ sk\end{array}$$

$$\begin{array}{r}\emptyset\\ p\end{array} \ -[t] \overset{\sigma\ (x\,:\,T)}{\longrightarrow}_k \ \langle \sigma(\lambda(x\ :\ T) \Rightarrow t) \mid \emptyset\rangle$$

$$\sigma_1(t_1)\ \cdots\ \sigma_s(t_s) \oplus \mathrm{Zcase}_T(\sigma(u_1\cdots u_s)/p),\ sk \ \overset{i}{\longrightarrow}_k\ k\ \emptyset\ u_i\ \sigma\ (\circ\ \sigma_1(t_1)\cdots\circ\ \sigma_s(t_s),\ sk)$$

$$\begin{array}{r}\sigma_1(t_1)\ \cdots\ \sigma_s(t_s)\oplus\\ \mathrm{Zfix}\ (j,\ \sigma(f : T := v)/p :: \sigma'_1(u_1)\cdots\sigma'_{j-1}(u_{j-1})),\\ sk\end{array} \ \overset{i}{\longrightarrow}_k\ \begin{array}{l}k\ \emptyset\ v\ (\left\{f := \mathrm{Sfix}_j\ \sigma(f : T := v)/p,\ \sigma\right\})\\ (\sigma'_1(u_1)\cdots\sigma'_{j-1}(u_{j-1}),\\ \mathrm{Zcstr}_i\ \sigma_1(t_1)\cdots\sigma_s(t_s),\ sk)\end{array}$$

$$\sigma_1(t_1)\ \cdots\ \sigma_s(t_s) \oplus \emptyset \ \overset{i}{\longrightarrow}_k\ \langle C_i \mid \sigma_1(t_1)\cdots\sigma_s(t_s),\ \emptyset\rangle$$

contenu calculatoire et qu'en général on ne souhaite pas déplier. Certaines constantes sont des applications partielles d'autres constantes que l'on ne souhaite déplier que si elles ont elles-même un nombre suffisant d'arguments pour poursuivre des calculs. Enfin les constantes représentant des point fixes construits non pas à partir d'une analyse de cas élémentaire sur leur argument récursif mais à l'aide d'un filtrage complexe ne doivent se déplier que si les arguments permettent d'atteindre une feuille de l'arbre de décision du filtrage et non pas uniquement si l'argument récursif commence par un constructeur. D'autres situations sont imaginables. Bref, il faut offrir à l'utilisateur de configurer pour chaque constante globale son comportement vis à vis du dépliage.

Notre machine peut en tout cas comprendre ces directives :

- Ne jamais se déplier, c'est ne pas se déplier !
- Ne pas se replier, c'est ne pas être mis dans la trace des constantes dépliées.
- Contrôler le nombre d'arguments, c'est compter le nombre de noeuds $\circ\ t$ en tête de pile.
- Nous savons déjà geler le dépliage d'un point fixe en attendant de savoir si son argument récursif se réduit sur un constructeur. Nous pouvons similairement ajouter le nécessaire pour réduire les arguments d'une constante afin de contrôler s'ils permettent d'atteindre une feuille d'un arbre de décision donné.
  En pratique, une table stockant les arbres de décision sous-jacents pour toutes les constantes est définie. Elle est interrogée lorsque le terme de l'état de la machine est une constante. Le premier argument sur lequel travaille l'arbre de décision est placé en tête de machine et on empile un noeud Zconst (cst, $arg_1 \cdots arg_s$, split_tree) qui stocke la constante, les arguments déjà traités et l'arbre de décision qui reste à parcourir. Le calcul dans le cas des constructeurs est enfin étendu pour gérer ces noeuds Zconst (cst, $arg_1 \cdots arg_s$, st). Si l'arbre de décision restant à considérer (st) est vide, la constante cst' est effectivement dépliée. Sinon, le prochain argument $args_k$ qu'il faut évaluer est mis en tête.

$$\boxed{\Delta \vdash_p \langle \sigma(tm_i) \mid sk_i \rangle \downarrow_{RK} \langle \sigma'(tm_o) \mid sk_o \rangle}$$

$$\frac{\Delta \vdash_{add_arg\ u\ \sigma\ p} \langle \sigma(t) \mid \circ\ \sigma(u),\ sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}{\Delta \vdash_p \langle \sigma(t\ u) \mid sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}$$

$$\frac{}{\Delta \vdash_p \langle \sigma(s) \mid \emptyset \rangle \downarrow_{RK} \langle \sigma(s) \mid \emptyset \rangle} \qquad \frac{}{\Delta \vdash_p \langle \sigma(\forall (x : S)\ ,\ T) \mid \emptyset \rangle \downarrow_{RK} \langle \sigma(\forall (x : S)\ ,\ T) \mid \emptyset \rangle}$$

$$\frac{\Delta \vdash_\emptyset \langle \sigma(t) \mid Zcase_T(\sigma(u_1 \cdots u_s)/p),\ sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}{\Delta \vdash_p \langle \sigma(\texttt{case}\ t\ \texttt{predicate}\ T\ \texttt{of}\ u_1 \cdots u_s\ \texttt{end}) \mid sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}$$

$$\frac{sk \xrightarrow{i}_k \langle \sigma'(v) \mid sk' \rangle}{\Delta \vdash_p \langle \sigma(C_i) \mid sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle} \qquad \frac{\genfrac{}{}{0pt}{}{sk}{p} -[t] \xrightarrow{\sigma\ (x : T)}_k \langle \sigma'(v) \mid sk' \rangle}{\Delta \vdash_p \langle \sigma(\lambda (x : T) \Rightarrow t) \mid sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}$$

$$\frac{\Delta \vdash_\emptyset \langle \sigma_i(u_i) \mid Zfix\ (i,\ \sigma(f : T := t)/p :: \sigma_1(u_1) \cdots \sigma_{i\text{-}1}(u_{i\text{-}1})),\ sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}{\Delta \vdash_p \langle \sigma(\mathsf{fix}_i\ (f : T := t)) \mid \circ\ \sigma_1(u_1) \cdots \circ\ \sigma_i(u_i),\ sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}$$

$$\frac{sk = \circ\ \sigma_1(u_1) \cdots \circ\ \sigma_{i\text{-}k}(u_{i\text{-}k}),\ \emptyset}{\Delta \vdash_p \langle \sigma(\mathsf{fix}_i\ (f : T := t)) \mid sk \rangle \downarrow_{RK} \langle \sigma(\mathsf{fix}_i\ (f : T := t)) \mid sk \rangle}$$

$$\frac{\Delta \vdash_p \langle \sigma(t) \mid sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}{\Delta \vdash_p \langle (\sigma'',\ \{x := Sapp\ \sigma(t),\ \sigma'''\})(x) \mid sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}$$

$$\frac{(\circ\ \sigma_1(u)_1 \cdots \circ\ \sigma_s(u)_s,\ sk) \xrightarrow{i}_k \langle \sigma'(v) \mid sk' \rangle}{\Delta \vdash_{p'} \langle (\sigma'',\ \{x := Scstr_i\ \sigma_1(u_1) \cdots \sigma_s(u_s),\ \sigma'''\})(x) \mid sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}$$

$$\frac{\Delta \vdash_p \left\langle \sigma(\mathsf{fix}_j\ (f : T := t)) \mid sk \right\rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}{\Delta \vdash_p \left\langle (\sigma'',\ \left\{x := Sfix_j\ \sigma(f : T := t)/p',\ \sigma'''\right\})(x) \mid sk \right\rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}$$

$$\frac{}{\Delta \vdash_p \langle \sigma(\mathbf{I}) \mid \circ\ \sigma_1(t)_1 \cdots \circ\ \sigma_s(t)_s,\ \emptyset \rangle \downarrow_{RK} \langle \sigma(\mathbf{I}) \mid \circ\ \sigma_1(t)_1 \cdots \circ\ \sigma_s(t)_s,\ \emptyset \rangle}$$

$$\frac{\Delta,\ (c : T := t),\ \Delta' \vdash_{add_cst\ c\ p} \langle \emptyset(t) \mid sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}{\Delta,\ (c : T := t),\ \Delta' \vdash_p \langle \sigma(c) \mid sk \rangle \downarrow_{RK} \langle \sigma'(v) \mid sk' \rangle}$$

Figure 5.7 – Refoldant Krivine Abstract Machine

5.5 Discussion

La configurabilité ne suffit pas à offrir toutes les possibilités. Dans notre implantation, certains choix sont codés en dur, d'autres sont impossibles. Nous les discutons ici.

Cascade de constante Il existe des systèmes logiques comme Agda [14] ou Matita [1] dont les points fixes sont « génératifs ». Les points fixes n'y sont pas des objets locaux mais des constantes globales capables de s'appeler elles-mêmes. Le problème des repliages est alors bien moins critique.

Par contre, dans le cas d'une constante définie comme list_map appliquée à une fonction particulière, exprimer les appels récursifs en fonction de cette constante plutôt que de l'itérateur est un gain. Notre système est donc plus qu'une simulation du cas génératif en présence de cascade de constantes.

Pour autant, la réponse apportée n'est pas idéale. Elle est extrêmement sensible à la syntaxe et suppose une expertise de l'utilisateur afin d'être exploitable. Un simple échange d'argument et le système est perdu. Par exemple, en définissant plus par récurrence sur son premier argument puis succ m par plus m (**S O**), l'information conservée au cours de la réduction de succ (**S** n) est que pl (**S** n) (**S O**) est repliable en succ (**S** n), ce qui est vrai mais inutile. Le résultat obtenu est alors seulement **S** (plus n (**S O**)).

Points fixes mutuels Nous n'en avons pas parlé car ils ne changeaient jusqu'ici rien à notre étude mais `Coq` offre la possibilité de définir des types de donnée et des points fixes mutuellement récursifs. La machine abstraite ne conserve que les noms des constantes qu'elle a dépliées au cours de la réduction. Il n'y a donc aucun moyen direct de connaître le nom de la constante globale qui correspond à un appel récursif à un autre point fixe du paquet !

Replier ou réduire Les constantes définies à partir de list_map sont toujours convenablement repliables car la fonction est un paramètre de la définition. Dans le cas de list_fold, par contre, l'accumulateur évolue et les appels récursifs ne s'expriment pas forcément en fonction de la première constante. Nous choisissons de privilégier la réduction (et donc le repliage utilisant list_fold) sur la conservation de la constante haut niveau. L'autre choix peut avoir des usages.

Contrôler le nombre de dépliages Supposons qu'un utilisateur ait un type impliquant plus n (**S** (**S O**)) qu'il souhaite exprimer en fonction de plus n (**S O**) et non n. Il ne peut ni dire de ne pas déplier plus ni se contenter d'en autoriser le dépliage. Il lui faut un moyen d'autoriser un unique dépliage. L'implantation effective permet ce comportement en raffinant le moment où elle autorise certaines réductions.

6 Etablir qu'un point fixe ne produit pas de calcul infini

Prenons le point fixe fix_1 (f : ∀(n : **nat**) , **False** := λ(n : **nat**) ⇒ f **O**). D'après les règles de la figure 1.9 section 1.4, si on oublie la condition Decr, ce terme du CCI est bien typé. Pourtant, f (**S O**), par exemple, est une preuve close de **False**. Ainsi, pour toute proposition logique P, il existe une preuve de P (cf figure 3.6). Cette preuve est `case` (fix_1 (f : ∀(n : **nat**) , **False** := f **O**)) (**S O**) `predicate` λ(_ : **False**) ⇒ P `of end`.

Le programmeur n'est pas plus satisfait que le mathématicien. Ce point fixe n'est pas autre chose qu'une boucle infinie qui ne répondra jamais de valeur. Toutes les garanties que nous souhaitons apporter sur les preuves/programmes s'écroulent donc si nous ne nous assurons pas que la réduction des point fixes termine. C'est justement le rôle du jugement qui demande, dans l'exemple, Decr(Δ,λ(n : **nat**) ⇒ f **O**,f,1) (La syntaxe qui nous allons introduire est $\Delta ;\{f := \text{STRICT}, \emptyset\} \#^1_f \langle \lambda(n : \textbf{nat}) \Rightarrow f\ \textbf{O} \mid \text{ARG}\rangle$). Son objectif est d'imposer la *terminaison par décroissance structurelle* de f suivant son 1$^{\text{er}}$ argument. Un point fixe dont le corps vérifie cette propriété est dit *gardé*.

Ce chapitre décrit comment le jugement de garde est défini. Partant du principe de base de décroissance structurelle, nous introduirons une à une quelques subtilités permettant de traiter un nombre toujours plus grand de cas de décroissance structurelle. Les premiers raffinements, qui concernent les branches impossibles ou les points-fixes imbriqués, existent dans Coq depuis le travail de Gimenez [23]. Nous introduirons et justifierons ensuite les nouveaux raffinements suivants : propagation des conditions de décroissance à travers les coupures entrelacées, propagation des conditions de décroissance à travers les $\beta\iota\phi$-rédex garantissant la réduction forte, prise en compte de questions d'efficacité.

6.1 Décroissance structurelle

Les points fixes ont été introduits en CCI afin de manipuler les structures de données récursives. Ils ne permettent que cela. La construction d'un point fixe est structurée autour d'un argument de type inductif explicitement nommé. Il est appelé *argument récursif*. L'argument i dans fix_i (f : T := t) est le numéro de l'argument récursif. Le principe de la garantie de terminaison des points fixes est :

> A chaque dépliage d'un point fixe, au moins un des constructeurs de tête de l'argument récursif est consommé

.

Par exemple, un point fixe f appliqué à **S** (**S O**) ne peut faire des appels récursifs qu'à f (**S O**) ou f **O**. Il consomme un ou deux constructeurs.

La description de l'algorithme qui assure que l'argument récursif d'un point fixe décroît structurellement repose sur les concepts énumérés ici.

La vérification de la propriété se base sur une classification des variables libres apparaissant dans la définition du corps du point fixe.

NO signifie que la variable n'est pas sous-terme de l'argument récursif.

STRICT signifie qu'elle est sous terme de l'argument récursif.

ARG qu'elle est l'argument récursif lui-même.

La catégorie à laquelle appartient chaque variable libre est stockée au sein d'un environnement que nous noterons Ψ. L'environnement Δ contient lui les définitions des constantes globales.

Deux phases distinctes se succèdent. D'abord

$\Delta ; \Psi \#^{i}_{f} \langle t \mid \pi \rangle$, le jugement principal, parcourt un terme t (le corps du point fixe) à la recherche des appels récursifs à f pour les vérifier. Cette tâche comporte deux aspects :
- Quand un lieur est traversé, le statut de sous-terme de la nouvelle variable libre introduite est déterminé et mémorisé dans l'environnement.
- Quand un appel récursif à f est atteint, son *i*-ième argument est contrôlé. Le second jugement vérifie que ce terme est sous-terme. Il utilise pour cela les valeurs de sous-terme des variables libres collectées dans l'environnement au fur et à mesure du parcours du corps du point fixe.

$\Delta ; \Psi \diamond \langle t \mid \pi \rangle = \mathrm{spec}$ donne la spécification de sous-terme spec d'un terme t (l'argument en position récursive d'un appel récursif) grâce aux spécifications de sous-terme de ses variables libres. Ce jugement est appelé par le premier à chaque fois qu'il atteint un appel récursif.

Calculer une valeur de sous-terme

Le dernier argument (π) des jugements est une pile. Comme une pile de machine abstraite, elle donne la valeur des variables correspondant aux lieurs en tête du terme à inspecter. La différence est que c'est ici une valeur de sous-terme et non un terme qui est associé à une variable.

En réalité, le deuxième jugement décrit figure 6.1 est effectivement une machine abstraite pour la β-réduction en appel par valeur. Il s'agit d'une machine qui calcule des spécifications de sous-terme.

Le premier jugement embarque cette machine abstraite afin de lancer des calculs avec un environnement correct.

Ces machines ont une règle particulière :

$$\frac{\Delta ; \Psi \diamond \langle u \mid \emptyset \rangle = c \qquad \Delta ; \Psi \diamond \langle t_1 \mid \mathtt{renforce}_1(c)_1 \cdots \mathtt{renforce}_1(c)_s \rangle = o_1 \qquad \cdots \qquad \Delta ; \Psi \diamond \langle t_n \mid \mathtt{renforce}_n(c)_1 \cdots \mathtt{renforce}_n(c)_s \rangle = o_n \qquad \prod_{k=1}^{n} o_k = o}{\Delta ; \Psi \diamond \langle \mathtt{case}\ u\ \mathtt{predicate}\ T\ \mathtt{of}\ t_1 \cdots t_n\ \mathtt{end} \mid \pi \rangle = o}$$

$$\boxed{\Delta\,;\Psi \diamond \langle t \mid \pi\rangle = out}$$

$$\frac{}{\Delta\,;\Psi \diamond \langle x \mid \pi\rangle = \Psi(x)} \qquad \frac{\Delta\,;\Psi \diamond \langle \Delta(c) \mid \pi\rangle = o}{\Delta\,;\Psi \diamond \langle c \mid \pi\rangle = o} \qquad \frac{}{\Delta\,;\Psi \diamond \langle s \mid \emptyset\rangle = NO}$$

$$\frac{\Delta\,;\{x := s,\, \Psi\} \diamond \langle t \mid \pi\rangle = out}{\Delta\,;\Psi \diamond \langle \lambda(x : S) \Rightarrow t \mid s,\, \pi\rangle = out} \qquad \frac{\Delta\,;\{x := NO,\, \Psi\} \diamond \langle t \mid \emptyset\rangle = out}{\Delta\,;\Psi \diamond \langle \lambda(x : S) \Rightarrow t \mid \emptyset\rangle = out}$$

$$\frac{\Delta\,;\Psi \diamond \langle u \mid \emptyset\rangle = s \qquad \Delta\,;\Psi \diamond \langle t \mid s,\, \pi\rangle = out}{\Delta\,;\Psi \diamond \langle t\ u \mid \pi\rangle = out} \qquad \frac{}{\Delta\,;\Psi \diamond \langle \forall(x : S)\,,\ T \mid \emptyset\rangle = NO}$$

$$\frac{}{\Delta\,;\Psi \diamond \langle \mathbf{C} \mid \pi\rangle = NO} \qquad \frac{}{\Delta\,;\Psi \diamond \langle \mathbf{I} \mid \pi\rangle = NO}$$

$$\frac{\Delta\,;\{g := NO,\, \Psi\} \diamond \langle t \mid NO_1 \cdots NO_k\rangle = o}{\Delta\,;\Psi \diamond \left\langle \mathsf{fix}_j\ (g : T := t) \mid s_1 \cdots s_k\right\rangle = o}$$

$$\frac{\begin{array}{c}\Delta\,;\Psi \diamond \langle u \mid \emptyset\rangle = c \qquad \Delta\,;\Psi \diamond \langle t_1 \mid \mathtt{renforce}_1(c)_1 \cdots \mathtt{renforce}_1(c)_s\rangle = o_1 \\ \cdots \qquad \Delta\,;\Psi \diamond \langle t_n \mid \mathtt{renforce}_n(c)_1 \cdots \mathtt{renforce}_n(c)_s\rangle = o_n \qquad \prod_{k=1}^{n} o_k = o\end{array}}{\Delta\,;\Psi \diamond \langle \mathtt{case}\ u\ \mathtt{predicate}\ T\ \mathtt{of}\ t_1 \cdots t_n\ \mathtt{end} \mid \pi\rangle = o}$$

FIGURE 6.1 – Calcul de la spécification de sous terme d'un terme

Cette règle est la clé de voûte de la garantie de la terminaison. Elle utilise deux fonctions sur les valeurs de sous-terme à expliquer.

— Nous souhaitons que n soit un sous-terme si t de type **nat** est l'argument récursif ou déjà un sous-terme dans `case` t `predicate` λ(m : **nat**) $\Rightarrow$ T `of` |**O**:a |**S**:λ(n : **nat**) $\Rightarrow$ b `end`.
Plus génériquement, la spécification STRICT est assignée aux variables représentant les arguments des constructeurs dans les branche lorsque le terme filtré a la spécification ARG. Pas à toutes les variables, si le terme filtré a le type **I** $t_1 \cdots t_s$, uniquement aux variables de type $\forall\overrightarrow{(x : S)}$, **I** $u_1 \cdots u_s$. Les autres ont la spécification NO.
En effet, le CCI autorise la définition de constructeurs dont le type est polymorphe comme par exemple
`Inductive` **I** : Set := | **C** :($\forall$(P : Set), P $\rightarrow$ P) $\rightarrow$ **I**.
La condition sur les types des arguments du constructeur est donc indispensable afin que ne puisse pas être admise la définition non terminante :
```
Definition Paradox : False :=
 (fix1 (ni : I → False := λ(i : I) ⇒
  case i predicate λ(_ : I) ⇒ False of
   |C:λ(f : ∀ (P : Set) , P → P) ⇒ ni (f _ i)
    end)) (C (λ(P : Set) (x : P) ⇒ x)).
```
Formellement, la description de la machine utilise une fonction qui donne la caractéristique de sous-terme des arguments d'un constructeur "du bon type" dans une branche en fonction de la caractéristique de sous-terme de l'objet filtré. Sa spécification est bien plus complexe que sa définition car il s'agit de dire
$\texttt{renforce}_i$(NO) = $NO_1 \cdots NO_s$ et
$\texttt{renforce}_i$(ARG) = $STRICT_1 \cdots STRICT_s$ pour les arguments de $\mathbf{C_i}$ non polymorphes ($NO_1 \cdots NO_s$ sinon).
$\texttt{renforce}_i$(STRICT) = $STRICT_1 \cdots STRICT_s$

— L'autre problématique est de fusionner les valeurs de sous-terme de chacune des branches afin de répondre la valeur de sous-terme de l'ensemble d'une analyse de cas.
C'est objet de l'opération $\times$ définie figure 6.2. Cette opération se doit d'être associative commutative afin de ne pas être influencée par l'ordre des branches. L'assistant de preuve de votre choix le garantit très facilement.
Il faut de plus une caractéristique pour l'analyse de cas à zéro cas. Un inductif à zéro cas ne pouvant jamais être construit dans un contexte cohérent, son élimination ne peut avoir lieu que dans des cas d'absurdité, c'est-à-dire dans du code mort. Nous introduisons donc une spécification de sous-terme DEAD qui caractérise du code mort et qui est l'élément absorbant de l'opération $\times$. Une analyse de cas qui analyse du code mort est nécessairement du code mort. Il est donc admissible de faire un appel récursif sur un terme de spécification DEAD. Il n'arrivera en effet pas ! De même, $\texttt{renforce}_i$(DEAD) = $DEAD_1 \cdots DEAD_s$, une analyse de cas de code mort a des branches qui sont du code mort donc jamais les arguments des constructeurs ne seront habités.

Il ne reste alors plus qu'à donner le premier jugement, celui qui fait le contrôle des appels récursifs, dans la figure 6.3. La garantie de la terminaison de l'évaluation de $\texttt{fix}_i$ (f : T := t) est

×	STRICT	ARG	DEAD	NO
NO	NO	NO	NO	NO
DEAD	STRICT	ARG	DEAD	
ARG	ARG	ARG		
STRICT	STRICT			

Figure 6.2 – Fusion de la valeur de sous termes de deux branches

le jugement $\Delta\,;\{f := \mathrm{NO}, \emptyset\} \#^{i}_{f} \langle t \mid \emptyset\rangle$.

La correction de cette condition de garde a été démontrée par Eduardo Gimenez [23] grâce à un algorithme qui, pour tout point fixe définissable dans ce système, définit un inductif dont le principe de récurrence simule le point fixe.

Retour sur les branches impossibles d'une analyse de cas

Nous sommes à présent en pleine mesure de comprendre la section 4.6. Si v est l'argument récursif, quelle est la valeur de sous-terme de Vhd v ?

L'astuce est que, grâce aux constructions que nous venons de faire, False_rect a la spécification de sous-terme DEAD. La branche impossible de **nil** n'est donc pas un handicap au fait que la spécification du code suivant soit bien STRICT :

```
case v predicate diag of
|nil:False_rect True |cons:λ(n : nat) (h : A) (t : vect A n) ⇒ h
end
```

6.2 Raffinement lié à l'induction

Dans le corps t d'un point fixe f peut apparaître la définition d'un autre point fixe g. On parle alors de points fixes imbriqués.

La garde des point fixes imbriqués n'est pas contrôlée simultanément. D'abord, t est typé. Comme la définition de g est dans t, le fait que g est gardé est contrôlé durant cette phase. Ensuite, on vérifie la terminaison de f et seulement de f.

Par contre, durant la vérification de f, g apparaît bel et bien. Il faut alors attribuer une valeur de sous-terme à g. Il faut de même attribuer une valeur de sous-terme à f. La prudence dicte que la spécification de sous-terme de f et g soit NO. De même, ce n'est pas parce que le i^e^ argument de g est un STRICT quand f appelle g que le i^e^ argument de g lors d'un appel récursif dans la définition de g restera un argument de spécification STRICT. Il est donc indispensable que tous les arguments de g aient la spécification NO.

Voilà justifiée la règle

$$\frac{\Delta\,;\Psi \#^{i}_{f} \langle T \mid \emptyset\rangle \qquad \Delta\,;\{g := \mathrm{NO}, \Psi\} \#^{i}_{f} \langle t \mid \mathrm{NO}_1 \cdots \mathrm{NO}_k\rangle}{\Delta\,;\Psi \#^{i}_{f} \left\langle \mathsf{fix}_j\ (g : T := t) \mid s_1 \cdots s_k\right\rangle}$$

et l'appel initial $\Delta\,;\{f := \mathrm{NO}, \emptyset\} \#^{i}_{f} \langle t \mid \emptyset\rangle$.

$$\boxed{\Delta\,;\Psi\ \#^{i}_{f}\ \langle t \mid \pi\rangle}$$

$$\frac{}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle x \mid \pi\rangle} \qquad \frac{}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle \mathbf{I} \mid \pi\rangle} \qquad \frac{\Delta\,;\Psi\ \#^{i}_{f}\ \langle S \mid \emptyset\rangle \qquad \Delta\,;\{x := s,\ \Psi\}\ \#^{i}_{f}\ \langle t \mid \pi\rangle}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle \lambda(x : S) \Rightarrow t \mid s,\ \pi\rangle}$$

$$\frac{}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle s \mid \emptyset\rangle} \qquad \frac{}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle \mathbf{C} \mid \pi\rangle} \qquad \frac{\Delta\,;\Psi\ \#^{i}_{f}\ \langle S \mid \emptyset\rangle \qquad \Delta\,;\{x := \mathrm{NO},\ \Psi\}\ \#^{i}_{f}\ \langle t \mid \emptyset\rangle}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle \lambda(x : S) \Rightarrow t \mid \emptyset\rangle}$$

$$\frac{\Delta\,;\Psi\ \#^{i}_{f}\ \langle u \mid \emptyset\rangle \qquad \Delta\,;\Psi \diamond \langle u \mid \emptyset\rangle = s \qquad \Delta\,;\Psi\ \#^{i}_{f}\ \langle t \mid s,\ \pi\rangle}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle t\ u \mid \pi\rangle} \qquad \frac{}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle c \mid \pi\rangle}$$

$$\frac{s_i = \mathrm{STRICT} \vee s_i = \mathrm{DEAD}}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle f \mid s_1 \cdots s_k\rangle} \qquad \frac{\Delta\,;\Psi\ \#^{i}_{f}\ \langle S \mid \emptyset\rangle \qquad \Delta\,;\{x := \mathrm{NO},\ \Psi\}\ \#^{i}_{f}\ \langle T \mid \emptyset\rangle}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle \forall(x : S),\ T \mid \emptyset\rangle}$$

$$\frac{\Delta\,;\Psi\ \#^{i}_{f}\ \langle T \mid \emptyset\rangle \qquad \Delta\,;\{g := \mathrm{NO},\ \Psi\}\ \#^{i}_{f}\ \langle t \mid \mathrm{NO}_1 \cdots \mathrm{NO}_k\rangle}{\Delta\,;\Psi\ \#^{i}_{f}\ \left\langle \mathtt{fix}_j\ (g : T := t) \mid s_1 \cdots s_k\right\rangle}$$

$$\frac{\begin{array}{c}\Delta\,;\Psi\ \#^{i}_{f}\ \langle u \mid \emptyset\rangle \qquad \Delta\,;\Psi\ \#^{i}_{f}\ \langle T \mid \emptyset\rangle \qquad \Delta\,;\Psi \diamond \langle u \mid \emptyset\rangle = c \\ \Delta\,;\Psi\ \#^{i}_{f}\ \langle t_1 \mid \mathtt{renforce}_1(c)\rangle \qquad \cdots \qquad \Delta\,;\Psi\ \#^{i}_{f}\ \langle t_n \mid \mathtt{renforce}_n(c)\rangle\end{array}}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle \mathtt{case}\ u\ \mathtt{predicate}\ T\ \mathtt{of}\ t_1 \cdots t_n\ \mathtt{end} \mid \pi\rangle}$$

FIGURE 6.3 – Contrôle de la décroissance structurelle de f selon son *i*-ème argument.

Pourtant, deux considérations peuvent améliorer l'expressivité :

1. Même si les arguments de g en général peuvent devenir complètement quelconques au fur et à mesure des appels récursifs, l'argument récursif a lui l'obligation de décroître. Ainsi, si g est appelé avec un argument STRICT en position récursive, cet argument ne peut que rapetisser lors d'appels récursifs et rester donc forcément STRICT.

   La pile des arguments peut ne pas être complètement réinitialisée ; l'argument récursif a le droit de garder sa valeur.

2. Les variables représentant les points fixes peuvent avoir la spécification STRICT. Il y a alors une forme de raisonnement par récurrence : Pour calculer si l'ensemble de la définition d'un point fixe est sous-terme, supposons que les appels récursifs qui apparaissent dans cette définition sont sous-terme.

La règle implémentée est donc

$$\frac{\Delta\,;\Psi\ \#^{i}_{f}\ \langle T \mid \emptyset\rangle \qquad \Delta\,;\{g := \text{STRICT},\ \Psi\}\ \#^{i}_{f}\ \left\langle t \mid NO_1 \cdots NO_{j\text{-}1},\ s_j,\ NO_1 \cdots NO_{k\text{-}j}\right\rangle}{\Delta\,;\Psi\ \#^{i}_{f}\ \left\langle \mathtt{fix}_j\ (g : T := t) \mid s_1 \cdots s_k\right\rangle}$$

et l'appel initial $\Delta\,;\{f := \text{STRICT}, \emptyset\}\ \#^{i}_{f}\ \langle t \mid \emptyset\rangle$.

Cette optimisation est présente dans les versions précédant la version 8.4 de `Coq`. La condition de garde jusque la version 8.3 de `Coq` est strictement composée des règles présentées jusqu'ici appliquées à la forme normale du corps du point fixe.

Nous abordons maintenant les raffinements postérieurs qui constituent la contribution originale de ce manuscrit.

6.3 Raffinement lié aux coupures entrelacées

Le chapitre 4 a montré que les coupures entrelacées étaient essentielles à l'écriture d'analyse de cas dépendant.

La contre-partie regrettable est que sont perdues en chemin les valeurs de sous-terme des arguments de l'analyse de cas. Avec les jugements des figures 6.3 et 6.1 $\mathtt{fix}_1\ (f : \ldots := \lambda(x\ :\ \mathbf{nat}) \Rightarrow \ldots \mathtt{case} \ldots \mathtt{predicate} \ldots \mathtt{of} \ldots |\lambda(\ldots : \ldots)\ (x' : \mathbf{nat}) \Rightarrow f\ (pred\ x') \ldots \mathtt{end}\ x \ldots)$ est refusé car x' n'est pas vu comme ARG.

Une solution de contournement est simple. La pile des arguments construite avant d'atteindre une analyse de cas est transmise à chacune des branches de cette analyse de cas. Néanmoins, cette manoeuvre est mal typée. Les branches impossibles peuvent donc ne pas être en adéquation avec leur pile. Un invariant ne tient plus. La pile en regard d'un produit ou d'une sorte n'est plus forcément vide.

La règle qui assure la terminaison d'une analyse de cas devient

$$\frac{\begin{array}{c}\Delta\,;\Psi\ \#^{i}_{f}\ \langle u \mid \emptyset\rangle \qquad \Delta\,;\Psi\ \#^{i}_{f}\ \langle T \mid \emptyset\rangle \qquad \Delta\,;\Psi \diamond \langle u \mid \emptyset\rangle = c \\ \Delta\,;\Psi\ \#^{i}_{f}\ \langle t_1 \mid \mathtt{renforce}_1(c), \pi\rangle \qquad \cdots \qquad \Delta\,;\Psi\ \#^{i}_{f}\ \langle t_n \mid \mathtt{renforce}_n(c), \pi\rangle\end{array}}{\Delta\,;\Psi\ \#^{i}_{f}\ \langle \mathtt{case}\ u\ \mathtt{predicate}\ T\ \mathtt{of}\ t_1 \cdots t_n\ \mathtt{end} \mid \pi\rangle}$$

Afin de gérer la terminaison des branches impossibles (et donc d'un type incompatible avec la pile qui leur est présentée), nous modifions les règles à propos des constructeurs de type (sorte et produit) pour qu'elles acceptent une pile :

$$\frac{}{\Delta ;\Psi \#^{i}_{f} \langle s \mid \pi \rangle}$$

6.4 Raffinement lié à la réduction

6.4.1 Garder la forme $\beta\iota\delta$ normale

Pour partager du code en utilisant par exemple des constantes globales, il arrive d'écrire des points fixes dont le corps contient des coupures. Or, la présence de coupure perturbe la vérification de la terminaison. Un exemple extrême est de définir la constante app qui applique une fonction à son argument. Si f est le point fixe que l'on contrôle, app f x n'est pas gardé car f n'est appliqué directement à aucun argument donc pas à un sous-terme alors que f x peut être gardé si x est STRICT.

Nous devons donc vérifier non pas le corps du point fixe mais la forme normale du corps du point fixe. Pour le faire, nous simulons les β, les ι et les ϕ réductions au sein de nos jugements.

Pour réaliser les ι réductions, le plus simple est d'ajouter une spécification de sous-terme qui signifie : être le i[e] constructeur dont les arguments ont les spécifications de sous-terme $s_1 \cdots s_n$. Pour cela la règle qui donne la spécification d'un constructeur est modifiée pour devenir

$$\frac{}{\Delta ;\Psi \diamond \left\langle C_j \mid c_1 \cdots c_s \right\rangle = \mathrm{CONSTR}_j(c_1 \cdots c_s)}$$

Parallèlement, la fonction `renforce` est enrichie d'une règle qui simule la ι-reduction en rendant leur spécification aux arguments du constructeur et rendant les autres branches code mort : $\texttt{renforce}_i(\mathrm{CONSTR}_j(o_1 \cdots o_s)) = o_1 \cdots o_s$ si $i = j$ et $\mathrm{DEAD}_1 \cdots \mathrm{DEAD}_s$ sinon.

Pour réaliser les β-réductions, nous allons retrouver une pile et un environnement de machine abstraite standard où des termes sont stockés. Les nœuds qui stockent un terme sont notées WEAK(Ψ,t). Ils seront aussi notés STRONG(Ψ,t) dans certains cas. Cette deuxième construction est la conséquence d'une optimisation qui va être détaillée à la fin de cette section. Le calcul des valeurs de sous-terme est identique dans les 2 cas. La vérification des appels récursifs pourrait être réalisée en n'utilisant que WEAK(Ψ,t).

Des nœuds "variables libres dont la spécification de sous-terme est s" restent nécessaires. Nous les notons $\top_s$.

La fonction $\texttt{renforce}_i(s)$ est elle aussi modifiée suite au changement de type des éléments de la pile. Si elle renvoyait $o_1 \cdots o_s$, elle renvoie maintenant $\top_{o1} \cdots \top_{os}$.

Puisque l'environnement stocke parfois des termes et non des valeurs de sous-terme directement, la fonction $\Delta \Downarrow \langle s \mid \pi \rangle = o$ présentée figure 6.4 permet de recalculer cette spécification de sous-terme.

La description exhaustive de la machine qui normalise un terme et en calcule sa spécification de sous-terme en appel par nom est donnée figure 6.5.

$$\boxed{\Delta \Downarrow \langle s \mid \pi \rangle = o}$$

$$\frac{}{\Delta \Downarrow \langle \top_s \mid \pi \rangle = s} \qquad \frac{\Delta ; \Psi \diamond \langle t \mid \pi \rangle = o}{\Delta \Downarrow \langle \mathrm{STRONG}(\Psi, t) \mid \pi \rangle = o} \qquad \frac{\Delta ; \Psi \diamond \langle t \mid \pi \rangle = o}{\Delta \Downarrow \langle \mathrm{WEAK}(\Psi, t) \mid \pi \rangle = o}$$

FIGURE 6.4 – Forcer le calcul d'une valeur de sous terme

$$\boxed{\Delta ; \Psi \diamond \langle t \mid \pi \rangle = out}$$

$$\frac{\Delta \Downarrow \langle \Psi(x) \mid \pi \rangle = out}{\Delta ; \Psi \diamond \langle x \mid \pi \rangle = out} \qquad \frac{\Delta ; \{x := s, \Psi\} \diamond \langle t \mid \pi \rangle = out}{\Delta ; \Psi \diamond \langle \lambda(x : S) \Rightarrow t \mid s, \pi \rangle = out}$$

$$\frac{\Delta ; \{x := \top_{NO}, \Psi\} \diamond \langle t \mid \emptyset \rangle = out}{\Delta ; \Psi \diamond \langle \lambda(x : S) \Rightarrow t \mid \emptyset \rangle = out} \qquad \frac{\Delta ; \Psi \diamond \langle u \mid \emptyset \rangle = c \qquad \Delta ; \Psi \diamond \langle t \mid \top_c, \pi \rangle = out}{\Delta ; \Psi \diamond \langle t\ u \mid \pi \rangle = out}$$

$$\frac{}{\Delta ; \Psi \diamond \langle s \mid \pi \rangle = NO} \qquad \frac{}{\Delta ; \Psi \diamond \langle \forall(x : S), T \mid \pi \rangle = NO}$$

$$\frac{\Delta \Downarrow \langle c_1 \mid \emptyset \rangle = o_1 \qquad \cdots \qquad \Delta \Downarrow \langle c_s \mid \emptyset \rangle = o_s}{\Delta ; \Psi \diamond \left\langle C_j \mid c_1 \cdots c_s \right\rangle = \mathrm{CONSTR}_j(o_1 \cdots o_s)} \qquad \frac{}{\Delta ; \Psi \diamond \langle \mathbf{I} \mid \pi \rangle = NO}$$

$$\frac{\Delta \Downarrow \left\langle s_j \mid \emptyset \right\rangle = \mathrm{CONSTR}_k(c_1 \cdots c_s) \qquad \Delta ; \left\{g := \mathrm{WEAK}(\Psi, \mathsf{fix}_j\ (g : T := t)), \Psi\right\} \diamond \langle t \mid s_1 \cdots s_k \rangle = o}{\Delta ; \Psi \diamond \left\langle \mathsf{fix}_j\ (g : T := t) \mid s_1 \cdots s_k \right\rangle = o} \qquad \frac{\Delta ; \Psi \diamond \langle \Delta(c) \mid \pi \rangle = o}{\Delta ; \Psi \diamond \langle c \mid \pi \rangle = o}$$

$$\frac{\Delta \Downarrow \left\langle s_j \mid \emptyset \right\rangle = o' \qquad \Delta ; \{g := \top_{STRICT}, \Psi\} \diamond \left\langle t \mid \top_{NO1} \cdots \top_{NOj\text{-}1}, \top_{o'}, \top_{NO1} \cdots \top_{NOk\text{-}j} \right\rangle = o}{\Delta ; \Psi \diamond \left\langle \mathsf{fix}_j\ (g : T := t) \mid s_1 \cdots s_k \right\rangle = o}$$

$$\frac{\Delta ; \Psi \diamond \langle u \mid \emptyset \rangle = c \qquad \Delta ; \Psi \diamond \langle t_1 \mid \mathtt{renforce}_1(c), \pi \rangle = o_1 \qquad \cdots \qquad \Delta ; \Psi \diamond \langle t_n \mid \mathtt{renforce}_n(c), \pi \rangle = o_n \qquad \prod_{k=1}^{n} o_k = o}{\Delta ; \Psi \diamond \langle \mathtt{case}\ u\ \mathtt{predicate}\ T\ \mathtt{of}\ t_1 \cdots t_n\ \mathtt{end} \mid \pi \rangle = o}$$

FIGURE 6.5 – Calcul de la spécification de sous terme d'un terme

Cette description donne une seconde règle pour le point fixe afin de réaliser la ϕ réduction en profitant de la même machinerie.

6.4.2 Retrouver la réduction forte

Vérifier que la forme normale du corps d'un point fixe vérifie la condition de garde est insuffisant à assurer la normalisation forte. Prenons l'exemple `Fixpoint` aie (n : **nat**) : **nat** := (λ(x : **nat**) $\Rightarrow$ **O**) (aie **O**)., la forme normale du corps : λ(n : **nat**) $\Rightarrow$ **O** est bien évidemment gardée. Son évaluation en appel par valeur, par contre, rentre dans une boucle infinie !

Il faut donc vérifier quand même les arguments des applications avant de les mettre dans la pile. La vérification à faire est un peu plus faible. Les arguments sont systématiquement évalués dans un contexte vide lors d'un calcul en appel par valeur. De ce fait, des variables libres en position d'argument récursif ne peuvent jamais déclencher de ϕ-réductions.

La règle qui vérifie une application est donc

$$\frac{\Delta ;\Psi \;\square^{i}_{f} \langle u \mid \emptyset\rangle \qquad \Delta ;\Psi \;\#^{i}_{f} \langle t \mid \mathrm{WEAK}(\Psi,u), \pi\rangle}{\Delta ;\Psi \;\#^{i}_{f} \langle t\ u \mid \pi\rangle}$$

Une nouvelle valeur de sous terme NEUTRAL est ajoutée qui stipule qu'une variable est une variable libre.

Les fonctions sur les valeurs de sous-terme sont enrichies par

$\mathtt{renforce}_i(\mathrm{NEUTRAL}) = \mathrm{NEUTRAL}_1 \cdots \mathrm{NEUTRAL}_s$

et la table du produit des valeurs de sous-terme est complétée par rapport à la figure 6.2 par

$\times$	$\mathrm{CONSTR}_i(c_1 \cdots c_s)$	NEUTRAL
NO	NO	NO
DEAD	$\mathrm{CONSTR}_i(c_1 \cdots c_s)$	NEUTRAL
ARG	NO*	ARG but WEAK
STRICT	NO*	NEUTRAL
NEUTRAL	$\mathrm{CONSTR}_i(c_1 \cdots c_s)$ but WEAK	NEUTRAL
$\mathrm{CONSTR}_j(c'_1 \cdots c'_s)$	$\mathrm{CONSTR}_i(c_1 \times c'_1 \cdots c_s \times c'_s)$	

Nous appelons garde faible le jugement $\Delta ;\Psi \;\square^{i}_{f} \langle u \mid \pi\rangle$. Il assure que t en position d'argument est gardé. Il est défini par les règles :

Vérifier faiblement les arguments des valeurs (constructeurs, inductifs, variables libres) n'est pas suffisant. Les jugements correspondants doivent contrôler "normalement" les corps des arguments qui ont été mis dans la pile.

6.4.3 Questions d'efficacité

Les règles de déduction du jugement de vérification de la garde présenté figure 6.6 ne sont pas dirigées par la syntaxe puisque les constantes globales et les applications apparaissent toutes deux dans deux jugements différents. L'introduction des éléments de pile

$$\boxed{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle t \mid \pi \rangle}$$

$$\frac{}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle x \mid \pi \rangle} \qquad \frac{}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle \mathbf{I} \mid \pi \rangle} \qquad \frac{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle S \mid \emptyset \rangle \qquad \Delta\,;\{x := s,\, \Psi\} \,\Box^{i}_{f}\, \langle t \mid \pi \rangle}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle \lambda(x : S) \Rightarrow t \mid s,\, \pi \rangle}$$

$$\frac{}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle \mathbf{C} \mid \pi \rangle} \qquad \frac{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle u \mid \emptyset \rangle \qquad \Delta\,;\Psi \,\Box^{i}_{f}\, \langle t \mid \mathrm{WEAK}(\Psi,u),\, \pi \rangle}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle t\ u \mid \pi \rangle}$$

$$\frac{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle S \mid \emptyset \rangle \qquad \Delta\,;\{x := \top_{\mathrm{NEUTRAL}},\, \Psi\} \,\Box^{i}_{f}\, \langle t \mid \emptyset \rangle}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle \lambda(x : S) \Rightarrow t \mid \emptyset \rangle} \qquad \frac{}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle s \mid \pi \rangle}$$

$$\frac{\Delta \Downarrow \langle s_i \mid \emptyset \rangle = o \qquad o = \mathrm{STRICT} \lor o = \mathrm{DEAD} \lor o = \mathrm{NEUTRAL}}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle f \mid s_1 \cdots s_k \rangle}$$

$$\frac{}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle c \mid \pi \rangle} \qquad \frac{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle S \mid \emptyset \rangle \qquad \Delta\,;\{x := \mathrm{NO},\, \Psi\} \,\Box^{i}_{f}\, \langle T \mid \emptyset \rangle}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle \forall(x : S),\ T \mid \pi \rangle}$$

$$\frac{\begin{array}{c}\Delta \Downarrow \left\langle s_j \mid \emptyset \right\rangle = \mathrm{CONSTR}_k(c_1 \cdots c_s) \\ \Delta\,;\left\{g := \mathrm{WEAK}(\Psi, \mathtt{fix}_j\ (g : T := t)),\, \Psi\right\} \,\Box^{i}_{f}\, \langle t \mid s_1 \cdots s_k \rangle\end{array}}{\Delta\,;\Psi \,\Box^{i}_{f}\, \left\langle \mathtt{fix}_j\ (g : T := t) \mid s_1 \cdots s_k \right\rangle}$$

$$\frac{\begin{array}{c}\Delta\,;\Psi \,\Box^{i}_{f}\, \langle T \mid \emptyset \rangle \qquad \Delta \Downarrow \left\langle s_j \mid \emptyset \right\rangle = o \\ \Delta\,;\{g := \top_{\mathrm{STRICT}},\, \Psi\} \,\#^{i}_{f}\, \left\langle t \mid \top_{\mathrm{NO1}} \cdots \top_{\mathrm{NOj\text{-}1}},\, \top_{o},\, \top_{\mathrm{NO1}} \cdots \top_{\mathrm{NOk\text{-}j}} \right\rangle\end{array}}{\Delta\,;\Psi \,\Box^{i}_{f}\, \left\langle \mathtt{fix}_j\ (g : T := t) \mid s_1 \cdots s_k \right\rangle}$$

$$\frac{\begin{array}{c}\Delta\,;\Psi \,\Box^{i}_{f}\, \langle u \mid \emptyset \rangle \qquad \Delta\,;\Psi \,\Box^{i}_{f}\, \langle T \mid \emptyset \rangle \qquad \Delta\,;\Psi \diamond \langle u \mid \emptyset \rangle = c \\ \Delta\,;\Psi \,\Box^{i}_{f}\, \langle t_1 \mid \mathtt{renforce}_1(c),\, \pi \rangle \qquad \cdots \qquad \Delta\,;\Psi \,\Box^{i}_{f}\, \langle t_n \mid \mathtt{renforce}_n(c),\, \pi \rangle\end{array}}{\Delta\,;\Psi \,\Box^{i}_{f}\, \langle \mathtt{case}\ u\ \mathtt{predicate}\ T\ \mathtt{of}\ t_1 \cdots t_n\ \mathtt{end} \mid \pi \rangle}$$

FIGURE 6.6 – Contrôle de la décroissance structurelle de f selon son *i*-ème argument.

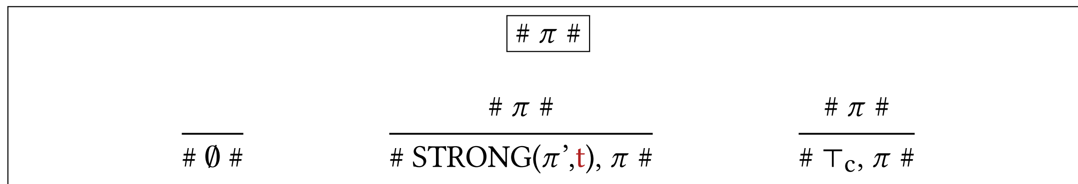


FIGURE 6.7 – Contrôle de la pile

STRONG(Ψ,t) n'a pas non plus été justifiée. Tout ceci découle d'une optimisation que nous expliquons maintenant.

Un argument dont on vérifie la garde faible mais qui ne fait jamais d'appels récursifs avec un variable libre en position d'argument récursif est fortement gardé. La garde faible peut ainsi répondre sans surcoût : le corps est en fait fortement gardé.

Cette information est précieuse car :

— un terme fortement gardé n'a pas à être revérifié à chacune de ses utilisations
— une constante globale dont tous les arguments sont fortement gardés n'a pas besoin d'être dépliée

Ainsi il n'existe pas réellement deux possibilités pour l'application en réalité mais une fonction de garde faible qui répond si la garde assurée est effectivement seulement faible ou de fait forte. L'élément inséré dans la pile reflète alors cette information.

Quand au non déterminisme à propos des constantes globales, il s'agit dans le code d'un dépliage paresseux.

Avec cette distinction dans la pile et l'environnement entre terme faiblement gardé et terme gardé, nous construisons une machine pour contrôler la terminaison par décroissance structurelle en appel par valeur quand c'est possible et en appel par nom quand c'est nécessaire.

Remarque Les jugements de la figure 6.6 souffrent d'une inefficacité pratique.

Pour calculer la spécification de sous-terme des variables représentant les argument d'un constructeur dans la branche d'une analyse de cas, il faut connaître la spécification de sous-terme du terme filtré. Or, ce calcul est potentiellement long et si aucune de ces variables n'est impliquée dans un appel récursif, il est inutile. Mieux vaut de ce fait calculer paresseusement les valeurs de sous-terme des variables libres.

6.5 La règle ϕ de réduction impose une garde structurelle

L'obligation de décroissance structurelle des points fixes est un carcan bien étroit. Certes, il y a encore de bien jolies mélodies à découvrir en Do Majeur [1] mais doit-on s'y contraindre ?

Malheureusement oui, dans le CCI tel qu'il est présenté ici. La règle de réduction ϕ nous interdit d'accepter un point fixe même terminant pour toutes entrées closes sans le vérifier selon nos critères.

Prenons le point fixe

1. Sergueï Prokofiev ? ? ? via Olivier Danvy

$$\boxed{\Delta\,;\Psi \,\#^{i}_{f}\, \langle t \mid \pi \rangle}$$

$$\frac{\#\ \pi\ \# \qquad \Psi(x) = \mathrm{STRONG}(\Psi',t) \lor \Psi(x) = \top_{c}}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle x \mid \pi \rangle} \qquad \frac{\#\ \pi\ \#}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle \mathbf{I} \mid \pi \rangle}$$

$$\frac{\begin{array}{c}\Psi(x) = \mathrm{STRONG}(\Psi',t) \lor \Psi(x) = \mathrm{WEAK}(\Psi',t)\\ \Delta\,;\Psi' \,\#^{i}_{f}\, \langle t \mid \pi \rangle\end{array}}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle x \mid \pi \rangle} \qquad \frac{\begin{array}{c}\Delta\,;\Psi \,\#^{i}_{f}\, \langle S \mid \emptyset \rangle\\ \Delta\,;\{x := \mathrm{NO}, \Psi\} \,\#^{i}_{f}\, \langle T \mid \emptyset \rangle\end{array}}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle \forall(x : S)\,,\ T \mid \pi \rangle}$$

$$\frac{\begin{array}{c}\Delta\,;\Psi \,\#^{i}_{f}\, \langle S \mid \emptyset \rangle\\ \Delta\,;\{x := s, \Psi\} \,\#^{i}_{f}\, \langle t \mid \pi \rangle\end{array}}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle \lambda(x : S) \Rightarrow t \mid s, \pi \rangle} \qquad \frac{}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle s \mid \pi \rangle} \qquad \frac{\begin{array}{c}\Delta\,;\Psi \,\#^{i}_{f}\, \langle S \mid \emptyset \rangle\\ \Delta\,;\{x := \top_{\mathrm{NO}}, \Psi\} \,\#^{i}_{f}\, \langle t \mid \emptyset \rangle\end{array}}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle \lambda(x : S) \Rightarrow t \mid \emptyset \rangle}$$

$$\frac{\#\ \pi\ \#}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle c \mid \pi \rangle} \qquad \frac{\begin{array}{c}\Delta\,;\Psi \,\#^{i}_{f}\, \langle u \mid \emptyset \rangle\\ \Delta\,;\Psi \,\#^{i}_{f}\, \langle t \mid \mathrm{STRONG}(\Psi,u), \pi \rangle\end{array}}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle t\ u \mid \pi \rangle} \qquad \frac{\Delta\,;\Psi \,\#^{i}_{f}\, \langle \Delta(c) \mid \pi \rangle}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle c \mid \pi \rangle}$$

$$\frac{\begin{array}{c}\Delta\,;\Psi \,\square^{i}_{f}\, \langle u \mid \emptyset \rangle\\ \Delta\,;\Psi \,\#^{i}_{f}\, \langle t \mid \mathrm{WEAK}(\Psi,u), \pi \rangle\end{array}}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle t\ u \mid \pi \rangle} \qquad \frac{\#\ \pi\ \#}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle \mathbf{C} \mid \pi \rangle} \qquad \frac{\begin{array}{c}\Delta \Downarrow \langle s_i \mid \emptyset \rangle = o\\ o = \mathrm{STRICT} \lor o = \mathrm{DEAD}\end{array}}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle f \mid s_1 \cdots s_k \rangle}$$

$$\frac{\begin{array}{c}\Delta \Downarrow \left\langle s_j \mid \emptyset \right\rangle = \mathrm{CONSTR}_k(c_1 \cdots c_s)\\ \Delta\,;\left\{g := \mathrm{WEAK}(\Psi, \mathsf{fix}_j\ (g : T := t)), \Psi\right\} \,\#^{i}_{f}\, \langle t \mid s_1 \cdots s_k \rangle\end{array}}{\Delta\,;\Psi \,\#^{i}_{f}\, \left\langle \mathsf{fix}_j\ (g : T := t) \mid s_1 \cdots s_k \right\rangle}$$

$$\frac{\begin{array}{c}\Delta\,;\Psi \,\#^{i}_{f}\, \langle T \mid \emptyset \rangle \qquad \Delta \Downarrow \left\langle s_j \mid \emptyset \right\rangle = o\\ \Delta\,;\{g := \top_{\mathrm{STRICT}}, \Psi\} \,\#^{i}_{f}\, \left\langle t \mid \top_{\mathrm{NO1}} \cdots \top_{\mathrm{NOj\text{-}1}}, \top_{o}, \top_{\mathrm{NO1}} \cdots \top_{\mathrm{NOk\text{-}j}} \right\rangle\end{array}}{\Delta\,;\Psi \,\#^{i}_{f}\, \left\langle \mathsf{fix}_j\ (g : T := t) \mid s_1 \cdots s_k \right\rangle}$$

$$\frac{\begin{array}{c}\Delta\,;\Psi \,\#^{i}_{f}\, \langle u \mid \emptyset \rangle \qquad \Delta\,;\Psi \,\#^{i}_{f}\, \langle T \mid \emptyset \rangle \qquad \Delta\,;\Psi \diamond \langle u \mid \emptyset \rangle = c\\ \Delta\,;\Psi \,\#^{i}_{f}\, \langle t_1 \mid \mathsf{renforce}_1(c), \pi \rangle \qquad \cdots \qquad \Delta\,;\Psi \,\#^{i}_{f}\, \langle t_n \mid \mathsf{renforce}_n(c), \pi \rangle\end{array}}{\Delta\,;\Psi \,\#^{i}_{f}\, \langle \mathsf{case}\ u\ \mathsf{predicate}\ T\ \mathsf{of}\ t_1 \cdots t_n\ \mathsf{end} \mid \pi \rangle}$$

FIGURE 6.8 – Contrôle de la décroissance structurelle de f selon son i-ème argument.

```
Fixpoint f (m : nat) (n : nat) : nat :=
 case m predicate λ(_ : nat) ⇒ nat of
 |S:λ(m' : nat) ⇒ f m' (S n)
 |O:case n predicate λ(_ : nat) ⇒ nat of
  |S:λ(n' : nat) ⇒ f O n'
  |O:O
  end
 end.
```

Pour tous terme clos, il se réduit vers **O**. Pourtant, ni m ni n ne décroissent structurellement. C'est (m, n) qui décroît pour un ordre lexicographique. Le point fixe f n'est pas définissable dans le CCI car il faudrait lui assigner un argument récursif qui en commanderait le dépliage. Or, si m est choisi, la réduction de λ(a : nat) ⇒ f **O** a diverge. Si n est choisi, la réduction de λ(a : nat) ⇒ f a **O** diverge.

Tout n'est pas perdu, avec du courage et l'aide des travaux de [8], il est possible de convertir une preuve de terminaison en un terme qui décroît structurellement à chaque appel et pouvoir ainsi écrire le point fixe voulu.

Conclusion et perspectives

La problématique transversale de ce manuscrit est d'étudier comment offrir au programmeur fonctionnel le meilleur environnement possible pour programmer avec des types dépendants. Les structures de données dépendantes, le calcul dans les types et la restriction à des programmes fortement normalisant augmentent d'autant la difficulté d'écrire un programme qu'elles améliorent les garanties apportées. La rigueur du vérificateur de type ne doit pourtant pas annihiler l'enthousiasme du programmeur. Pour ce faire, il faut mettre entre les deux des outils d'interaction. Notre tentative n'est qu'une étape sur la longue route que représente cette démarche.

Au delà de fournir un algorithme de génération des analyses de cas aux vertus essentiellement pratiques, ce manuscrit suggère très fortement une délimitation des analyses de cas nécessitant l'axiome K. Pour que K soit nécessaire au cours de l'élimination de u, il faut soit que nous n'ayons pas à faire à un filtrage hors du cadre délimité par Coquand : dans le type de u, il existe un argument de l'inductif instantié par un terme t ne pouvant pas être vu comme un motif. Il est sinon nécessaire que

1. la signature de l'inductif auquel appartient u contienne un argument x et un paramètre ou argument y de même type T,
2. x et y n'aient pas de puits commun dans le graphe dirigé des dépendances entre les paramètres et arguments de l'inductif,
3. dans le type de u, x et y soient instantiées par le même terme t,
4. le terme t apparaisse dans le type P attendu en retour du filtrage,
5. il n'existe pas de terme bien typé u dans lequel t n'apparaît pas tel que $(\lambda(x\ y\ :\ T) \Rightarrow u)\ t\ t$ soit convertible à P,
6. K ne soit pas vrai par ailleurs pour le type T.

La théorie homotopique des types offre un nouveau regard et appelle à poursuivre les travaux autour de l'égalité. Ce traitement pratique de la question par encodage est l'une des pistes pour améliorer la compréhension de cette problématique.

Ce manuscrit montre ensuite comment simuler une réduction nominale des points fixes dans un langage avec des points fixes structurels anonymes. Les réductions souhaitées pour simplifier un type sont un sous ensemble des réductions possibles. Les réductions évitées ne semblent pas être intéressantes non plus pour la conversion. Typiquement, dans l'échantillon des preuves utilisées comme tests, la convertibilité entre une constante globale représentant un point fixe appliqué à des arguments ne permettant pas d'atteindre une feuille de l'arbre de décision sous jacent au point fixe et sa forme réduite n'est jamais testé. Qu'en serait-il si la réduction primitive ne dépliait une constante que si le filtrage sous-jacent allait être complètement réduit ? Cette considération ainsi que les remarques sur la représentation des

constructeurs/analyse de cas du CCI argumentent pour rapprocher l'implantation machine de `Coq` de celle des langages fonctionnels efficaces. L'assistant de preuve se rapproche ainsi encore un peu plus d'un langage de programmation effectif.

Le dernier chapitre du manuscrit peut paraître désuet. Quel intérêt de contrôler une garde structurelle quand des théories de garantie de terminaison plus fines ont vu le jour. La réponse est double. D'abord, la machine de réduction du CCI est accrochée à la présence d'un argument récursif qui autorise les dépliages des points fixes. Il faudra donc revoir le dépliage des points fixes avant d'avoir recours à des critères de terminaison plus sophistiqués. Ensuite, avoir un vérificateur puissant mais demandant des annotations rend l'écriture des termes fastidieuse dans les cas trivialement corrects. Un système expressif vient donc quasiment systématiquement avec un algorithme heuristique pour inférer les annotations dans les cas simples. Il est alors intéressant de relire ce chapitre comme une heuristique intelligente pour deviner automatiquement les marques de taille dans les cas de décroissance structurelle.

D'un point de vue plus global, l'interaction homme-machine dans le domaine de la programmation avec type dépendant a été légèrement délaissée au profit de recherche en vue d'une amélioration des capacités de la machine. De grands résultats mathématiques ont été vérifiés par ordinateur. Il est inimaginable qu'ils l'aient été automatiquement. Faciliter l'interaction du vérificateur avec l'utilisateur a été la clé de ces réussites. Les travaux dans ce sens (par exemple [25, 12]) permettent aux assistants de preuves de rentrer progressivement dans la boite à outils du mathématicien. Il devrait en être de même pour le développeur. Améliorer les environnements de programmation grâce à la programmation dépendante est un enjeu à relever maintenant.

Bibliographie

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The matita interactive theorem prover. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer, 2011.

[2] Lennart Augustsson. Compiling pattern matching. In *FPCA*, pages 368–381, 1985.

[3] Lennart Augustsson. Cayenne - a language with dependent types. In *Advanced Functional Programming*, pages 240–267, 1998.

[4] Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2) :125–154, 1991.

[5] Bruno Barras. Verification of the interface of a small proof system in coq. In Eduardo Giménez and Christine Paulin-Mohring, editors, *TYPES*, volume 1512 of *Lecture Notes in Computer Science*, pages 28–45. Springer, 1996.

[6] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université Paris-Diderot—Paris VII, 1999.

[7] Bruno Barras, Pierre Corbineau, Benjamin Grégoire, Hugo Herbelin, and Jorge Luis Sacchini. A new elimination rule for the calculus of inductive constructions. In Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors, *TYPES*, volume 5497 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2008.

[8] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions : A practical tool for the coq proof assistant. In Masami Hagiya and Philip Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2006.

[9] Alessandro Berarducci and Corrado Böhm. General recursion on second order term algebras. In Aart Middeldorp, editor, *RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 2001.

[10] Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors. *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.

[11] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art :the Calculus of Inductive Constructions.* Springer-Verlag, 2004.

[12] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *J. Symb. Comput.*, 25(2) :161–194, 1998.

[13] Mathieu Boespflug. Conversion by evaluation. In Manuel Carro and Ricardo Peña, editors, *PADL*, volume 5937 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2010.

[14] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda - a functional language with dependent types. In Berghofer et al. [10], pages 73–78.

[15] Rod M. Burstall, David B. MacQueen, and Donald Sannella. Hope : An experimental applicative language. In *LISP Conference*, pages 136–143, 1980.

[16] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3) :95–120, 1988.

[17] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

[18] Cristina Cornes. *Conception d'un langage de haut niveau de representation de preuves.* PhD thesis, UNIVERSITE PARIS 7, November 1997.

[19] Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in coq. In Stefano Berardi and Mario Coppo, editors, *TYPES*, volume 1158 of *Lecture Notes in Computer Science*, pages 85–104. Springer, 1995.

[20] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007.

[21] David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[22] Peter Dybjer. Representing inductively defined sets by wellorderings in martin-löf's type theory. *Theor. Comput. Sci.*, 176(1-2) :329–335, 1997.

[23] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *TYPES*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

[24] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

[25] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008.

[26] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *ICFP*, pages 235–246. ACM, 2002.

[27] John Harrison. Hol light : An overview. In Berghofer et al. [10], pages 60–66.

[28] Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *LICS*, pages 208–212. IEEE Computer Society, 1994.

[29] Gérard Huet. Unification in typed lambda calculus. In C. Böhm, editor, *λ-Calculus and Computer Science Theory*, volume 37 of *Lecture Notes in Computer Science*, pages 192–212. Springer Berlin Heidelberg, 1975.

[30] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[31] Simon L. Peyton Jones. Haskell 98 : Expressions. *J. Funct. Program.*, 13(1) :17–38, 2003.

[32] Matt Kaufmann and J. Strother Moore. Enhancements to acl2 in versions 5.0, 6.0, and 6.1. In Ruben Gamboa and Jared Davis, editors, *ACL2*, volume 114 of *EPTCS*, pages 5–12, 2013.

[33] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3) :199–207, 2007.

[34] Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

[35] Gyesik Lee and Benjamin Werner. Proof-irrelevant model of cc with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 7(4), 2011.

[36] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 102(2) :177–207, 2010.

[37] Zhaohui Luo. Program specification and data refinement in type theory. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT, Vol.1*, volume 493 of *Lecture Notes in Computer Science*, pages 143–168. Springer, 1991.

[38] Luc Maranget. Compiling pattern matching to good decision trees. In Eijiro Sumii, editor, *ML*, pages 35–46. ACM, 2008.

[39] Conor McBride. Epigram : Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

[40] Conor McBride, Healfdene Goguen, and James McKinna. A few constructions on constructors. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2004.

[41] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1) :69–111, 2004.

[42] Alexandre Miquel and Benjamin Werner. The not so simple proof-irrelevant model of cc. In Herman Geuvers and Freek Wiedijk, editors, *TYPES*, volume 2646 of *Lecture Notes in Computer Science*, pages 240–258. Springer, 2002.

[43] Jean-François Monin and Xiaomu Shi. Handcrafted inversions made operational on operational semantics. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2013.

[44] Bengt Nordstrom, Kent Petersson, and Gordon Plotkin, editors. *WORKSHOP ON TYPES FOR PROOFS AND PROGRAMS Bastad*, 1992.

[45] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[46] Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

[47] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur.* Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[48] Frank Pfenning. Logical frameworks. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1063–1147. Elsevier and MIT Press, 2001.

[49] Frank Pfenning and Carsten Schürmann. System description : Twelf - a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer, 1999.

[50] Brigitte Pientka and Joshua Dunfield. Beluga : A framework for programming and reasoning with deductive systems (system description). In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21. Springer, 2010.

[51] Robert Pollack. *The Theory of LEGO : A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, Univ. of Edinburgh, 1994.

[52] The Univalent Foundations Program. Homotopy type theory : Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[53] Matthieu Sozeau. Equations : A dependent pattern-matching compiler. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2010.

[54] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer, 2008.

De nouveaux outils pour calculer avec des inductifs en `Coq`

En ajoutant au λ-calcul des structures de données algébriques, des types dépendants et un système de modules, on obtient un langage de programmation avec peu de primitives mais une très grande expressivité. L'assistant de preuve `Coq` s'appuie sur un tel langage (le CCI) à la sémantique particulièrement claire. L'utilisateur n'écrit pas directement de programme en CCI car cela est ardu et fastidieux. `Coq` propose un environnement de programmation qui facilite la tâche en permettant d'écrire des programmes incrémentalement grâce à des constructions de haut niveau plus concises.

Typiquement, les types dépendants imposent des contraintes fortes sur les données. Une analyse de cas peut n'avoir à traiter qu'un sous-ensemble des constructeurs d'un type algébrique, les autres étant impossibles par typage. Le type attendu dans chacun des cas varie en fonction du constructeur considéré. L'impossibilité de cas et les transformations de type doivent être explicitement écrites dans les termes du CCI. Pourtant, ce traitement est mécanisable et cette thèse décrit un algorithme pour réaliser cette automatisation.

Du point de vue du retour d'information de la part du système, il est nécessaire à l'interaction avec l'utilisateur de calculer des programmes du CIC sans faire exploser la taille syntaxique de la forme réduite. Cette thèse présente une machine abstraite conçu dans ce but.

Enfin, les points fixes permettent une manipulation aisée des structure de données récursives. En contrepartie, il faut s'assurer que leur exécution termine systématiquement. Cette question sensible fait l'objet du dernier chapitre de cette thèse.

New tools to compute with algebraic datatypes in `Coq`

The dependently typed λ-calculus with algebraic datastructures is a programming language with very few primitives but a huge expressivity. The `Coq` proof assistant is built over one variant of this language, the CIC. Its semantics is extremely clear but it is verbose. Therefore, users do not write programs directly in CIC. Instead, `Coq` provides tools to elaborate programs incrementally using higher level constructions.

Especially, mixing algebraic and dependent types increases the power and the difficulty of case analysis. Each case has a different type depending of the type of the constructor. Some cases are even impossible because of typing. These type casts and impossibility witnesses are explicit in CIC terms but they can be built mechanically. This thesis gives an algorithm to achieve this automation.

As far as feedback from the system is concerned, interaction with human asks for a way to compute `Coq` programs without making their syntactical length explode. This thesis propose a new abstract machine designed for this purpose.

Fixpoints provide a convenient way to deal with recursive datastructures. Nevertheless, ensuring their computation does not diverge on any entry is a challenging issue. It is tackled by the last chapter of this thesis.