Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq

Pierre Letouzey

N^o d'ordre : 7567

Thèse de doctorat

présentée à

L'Université de Paris-Sud

U.F.R. Scientifique d'Orsay

par

Pierre Letouzey

pour obtenir

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

Sujet :

Programmation fonctionnelle certifiée

L'extraction de programmes dans l'assistant Coq

Soutenue le 9 juillet 2004 devant la commission d'examen composée de

M.	Leroy Xavier	président
M.	Berardi Stefano	rapporteurs
M.	Monin Jean-François	
Mme	Benzaken Véronique	examinateurs
M.	Schwichtenberg Helmut	
Mme	Paulin Christine	directeur

Programmation fonctionnelle certifiée

L'extraction de programmes dans l'assistant Coq

Pierre Letouzey

Juillet 2004

Sommaire

Remerciements

Au moment d'apporter les dernières retouches à ce manuscrit, l'envie me vient de remercier la terre entière. Comme cela risquerait de rallonger un document déjà fort volumineux, je vais tenter d'être un peu plus sélectif, au risque d'en oublier certains. Que ceux-ci veuillent bien pardonner mon ingratitude.

En premier lieu, je voudrais exprimer ma profonde reconnaissance envers Christine Paulin. Durant toutes ces années, son encadrement a réellement été exemplaire. J'ai en permanence été impressionné par sa compétence, sa patience et sa gentillesse. Enfin, elle a toujours su rester à mon écoute malgré ses nombreuses obligations.

Je tiens ensuite à remercier chaleureusement les cinq autres membres du jury. Je suis très honoré qu'ils aient accepté de se pencher sur mon travail. Merci tout d'abord à Xavier Leroy : après m'avoir eu comme étudiant de DEA, le voila maintenant président de mon jury. Merci également aux deux rapporteurs Stefano Berardi et Jean-François Monin pour leurs relectures attentives et leurs commentaires constructifs. Merci ensuite à Véronique Benzaken, qui a bien voulu s'écarter ici de son domaine de prédilection. Merci enfin à Helmut Schwichtenberg d'avoir accepter de prolonger nos discussions sur l'extraction par cette participation à mon jury.

Ma gratitude va également à tous les membres des équipes de recherche dont je fais ou ai fait parti. Chacun a contribué à former un environnement de travail exceptionnel, particulièrement agréable et stimulant. Cela concerne bien sûr l'équipe Démons du LRI, avec dans le désordre les Jean-Christophe, Ralf, Claude, Evelyne, Judicaël, Xavier, Julien, Sylvain, Pierre, Nicolas, Benjamin, Jacek, Daria, Laurence, Jean-Pierre, Délia et Frédéric. Vient ensuite l'équipe Logical de l'INRIA Rocquencourt, autour de personnes comme Hugo Herbelin et Bruno Barras. De façon plus large, j'aimerai saluer les membres du LRI avec qui j'ai eu l'occasion de travailler, avec une mention spéciale pour Marwan Burelle. Enfin, je n'oublie pas l'équipe où j'ai fait mes premières armes en Coq, à savoir l'équipe Croap (devenue Lemme) de l'INRIA Sophia-Antipolis : un grand merci aux Laurent, Yves, Laurence, Francis, Loïc et leurs collègues.

Outre les membres de ces équipes, j'aimerais aussi remercier les chercheurs ayant essuyé les plâtres de ma nouvelle implantation de l'extraction Coq. Qu'il s'agisse de simples rapports de bogues, de demandes de fonctionnalités, ou bien de discussions plus poussées, cette nouvelle extraction leur doit beaucoup. Je pense ici par exemple à des membres de FT-R&D à Lannion, comme Jean-François Monin, Laurent Gottely, et Cuihtlauac Alvarado. Je suis

également profondément redevable au groupe de Henk Barendregt à Nimègue, en particulier à Milad Niqui, Bas Spitters et Luís Cruz-Filipe.

Dans la préface de sa thèse, Jean-Christophe Filliâtre remerciait en particulier certaines figures de l'informatique libre : Linus Torvalds pour Linux, Richard Stallman pour Emacs, Donald Knuth pour TEX, et Xavier Leroy pour Objective Caml. Le moins que je puisse faire est de m'associer à ces remerciements. Comment ferais-je en effet sans ces outils ? Je rajoute juste ici un immense merci à Jean-Christophe pour m'avoir familiarisé avec quantité de tels outils, et fourni un modèle LATEX pour ce document. À ce propos, merci également à Vincent Zoonekynd, qui est à l'origine du style des boîtes encadrant les exemples de ce manuscrit.

Si l'on remonte un peut plus dans le temps, un certain nombre de personnes ont eu une influence déterminante sur mon parcours jusqu'à cette thèse. J'aimerais saluer ici un certain nombre de mes anciens professeurs de mathématiques et d'informatique : Jean-Luc Ybert, André Lehault, Guy Méheut, Hervé Gianella, Jacques Chevallet, Laurent Chéno et Roberto Di Cosmo. Merci en particulier à Laurent Chéno de m'avoir fait découvrir à la fois le langage Caml et la notion de preuves de programmes.

Pour terminer, merci à tous mes amis, les Yann, Sandra, Gaël, Nathaëlle, Mylène, Nicolas, Eric, Mathilde, Louis, Lionel, Dimitri, Barbara et autres, pour leur bonne humeur, leurs grains de folie, et tous leurs plans « foireux, mais seulement à moitié ». Merci infiniment à ma famille et surtout à mes parents. Que serais-je sans eux ? Ma gratitude est bien difficile à traduire en mots. Je souhaite par ailleurs bon vent à ma soeur Catherine, tout juste agrégée. Enfin, merci à Antigone pour tout, et plus encore.

Introduction

Le besoin de programmes certifiés

Aujourd'hui, c'est une banalité de dire que les logiciels occupent dans nos sociétés modernes une place prépondérante, y compris dans des rôles critiques. La liste de ces missions de confiance désormais remplies par des programmes s'allonge sans cesse. Il peut s'agir bien sûr du contrôle d'équipements à risques comme des avions ou des centrales nucléaires, mais aussi d'opérations plus prosaïques comme la gestion de paiements électroniques. En tout cas, qu'il s'agisse de vies humaines ou d'argent, les enjeux sont énormes.

Malheureusement, c'est également une banalité que de constater la perfectibilité de ces programmes qui nous entourent. Sans pour autant céder au catastrophisme, force est de constater que des défaillances de logiciels font régulièrement les gros titres. L'exemple le plus fréquemment mentionné reste l'explosion de la fusée Ariane 5 en 1996. Plus près de nous, on peut citer la mise en évidence par S. Humpich de la vulnérabilité du système des cartes bancaires. Tout aussi alarmant, l'éditeur dominant actuellement le marché grand-public propose des logiciels qui sont affectés par des failles de sécurité à répétition, ce qui ouvre la porte à toutes sortes de virus, vers ou trojans. Voici enfin un témoignage lu dans un forum de discussion, répondant à une intervention opposant la faible qualité des logiciels pour PC à la grande fiabilité de ceux destinés à l'aéronautique :

> *J'apporterai toutefois une petite nuance quant à ton parallèle avec les avions de ligne. Leur système informatique n'est pas aussi solide que tu le dis, j'en sais quelque chose, je suis pilote sur A320 ! En fait on a régulièrement un ou plusieurs calculateurs qui plantent. C'est même normal, le soft est régulièrement modifié pour répondre aux nouvelles exigences réglementaires ou opérationnelles, de même que le hardware, sans compter que le tout doit subir l'assaut de dizaines de transferts électriques par jour ! Heureusement, tous les systèmes sont doublés voire triplés et même quintuplés dans certains cas ce qui fait que pendant que l'un redémarre tranquillement et effectue ses autotests, l'autre prend le relais, et la sécurité n'est jamais mise en jeu ...*

Là non plus, la situation n'est donc pas parfaite[1], la solution empirique de la redondance n'étant pas à toute épreuve.

Cette situation pratique insatisfaisante contraste de façon saisissante avec le point de vue des scientifiques. « L'informatique est une science exacte », écrivait en 1969 C. A. R. Hoare

[1] même si, en pratique, la très grande majorité des accidents d'avion sont dus à des erreurs humaines...

[45]. Ce dernier et d'autres comme R. W. Floyd ou E. W. Dijkstra ont ainsi donné un cadre mathématique à la programmation, précisant dans les années 1970 une notion de preuve de programme, aussi rigoureuse qu'une preuve de théorème. Hélas, trop souvent encore la programmation est abordée comme une méthode expérimentale : répétitions d'essais, de tests, d'erreurs et de corrections. Il est vrai qu'établir formellement la correction d'un programme demande actuellement des efforts très importants, alors qu'il est souvent très facile d'effectuer des tests. Mais ces tests peuvent s'avérer coûteux au final, et il est également bien rare qu'ils puissent être exhaustifs...

De toute façon, que l'on veuille certifier un programme ou simplement le tester, la première étape est d'élaborer une spécification, décrivant le comportement attendu de ce programme. Le résultat mathématique à prouver dans le cas d'une certification est alors que le programme satisfait bien sa spécification. Évidemment cette étape de spécification est un moment clé : si la spécification est incomplète ou incorrecte, rien n'empêchera un programme certifié correct vis-à-vis de cette spécification de mal se comporter lors de son exécution.

Les travaux de C. A. R. Hoare, R. W. Floyd et E. W. Dijkstra que nous venons d'évoquer sont originellement destinés à des programmes impératifs. Ainsi la logique de Hoare est-elle composée d'assertions portant sur les valeurs des variables du programme. Et la spécification d'une portion de code `C` est de la forme `{P}C{Q}` : si la pré-condition `P` est valide avant l'exécution du code `C`, alors la post-condition `Q` sera valide après l'action de `C` sur le contenu des variables. De même les outils de certification formelle les plus répandus dans l'industrie, comme la méthode B [1], sont axés sur les preuves de programmes impératifs. Ceci s'explique sans doute simplement par l'omniprésence du paradigme impératif dans l'industrie, au détriment des langages fonctionnels. La principale exception à ce jour est la création et l'usage d'Erlang chez Ericsson. Néanmoins, nous allons maintenant voir que les langages fonctionnels sont particulièrement adaptés à la création de programmes certifiés.

L'isomorphisme de Curry-Howard

Tous les langages fonctionnels possèdent comme noyau théorique commun le λ-calcul, que ce soit Lisp, Scheme, les membres de la famille ML comme Ocaml, ou encore Haskell. Or dans [26], H. Curry remarque dès 1958 que dans le λ-calcul simplement typé, tout terme bien typé a pour type une tautologie en logique propositionnelle intuitionniste[2], si l'on assimile type fonctionnel `A→B` et implication `A⇒B`.

Outre cet isomorphisme entre types et propositions, H. Curry établit également la correspondance entre termes et démonstrations. En particulier, la preuve constructive d'une implication `A⇒B` est une méthode qui permet de transformer toute démonstration `a` de la propriété `A` en une preuve `b` de la propriété `B`. Via l'isomorphisme, cette preuve de `A⇒B` peut donc être vue comme une fonction qui à tout objet `a` de type `A` associe un objet `b` de type `B`.

En 1969, W. A. Howard propose un λ-calcul avec types dépendants étendant l'isomorphisme à la logique intuitionniste du premier ordre [46]. Ainsi par exemple, une preuve constructive `∃x,P(x)` stipulant l'existence d'un objet vérifiant la propriété `P`, va donner via cet isomorphisme un programme fonctionnel construisant effectivement cet objet. Plus pré-

[2]Pour plus de détails sur les logiques intuitionnistes et constructives, on pourra consulter [10] et [80].

cisément, ce programme va retourner un couple `(x,p)` dans lequel `x` est l'objet recherché, et `p` un certificat démontrant que l'on a bien `P(x)`.

Par la suite, cet isomorphisme a été étendu à toutes sortes de systèmes logiques intuitionnistes à l'expressivité croissante, comme par exemple la théorie des types de Martin-Löf [58] ou encore le Calcul des Constructions Inductives (CCI) qui est à la base de l'assistant de preuve Coq [78]. Et l'extraction, c'est-à-dire cette possibilité de dériver un programme à partir d'une preuve, a été mis en pratique dans de nombreux systèmes, comme PX [44], Nuprl [50], Coq, Minlog [30] ou encore plus récemment Isabelle [13, 14].

On peut ici faire une comparaison avec la méthode de Hoare. Si l'on souhaite construire un programme certifié prenant en entrée un objet `x` vérifiant une certaine pré-condition `P(x)` et retournant alors en sortie un autre objet `y` vérifiant une post-condition `Q(x,y)`, il nous suffit alors de prouver dans un formalisme intuitionniste la proposition $\forall$`x,P(x)`$\rightarrow\exists$`y,Q(x,y)`. L'isomorphisme de Curry-Howard nous permet alors de dériver automatiquement de notre preuve un programme fonctionnel correct par construction.

En fait, le programme ainsi obtenu par un usage direct de cet isomorphisme n'est pas tout à fait celui espéré. Au lieu d'un programme prenant un `x` et retournant un `y`, on obtient plutôt un programme à deux arguments `x` et `p` et à deux résultats `y` et `q`. Et `p` et `q` sont alors deux certificats correspondants respectivement à des preuves de `P(x)` et `Q(x,y)`. Le rôle de l'extraction est alors de générer le programme naturel, *i.e.* sans certificats logiques, et de justifier cette suppression des certificats. Nous allons tout d'abord voir comment cela est fait en Coq, puis nous évoquerons ensuite le cas des autres assistants de preuve.

L'extraction en Coq

Cette thèse a donc été consacrée à l'étude de l'extraction en Coq. En fait, elle aurait pu s'intituler « Quinze ans d'extraction en Coq », puisqu'elle intervient quinze ans après une première thèse sur ce même sujet par C. Paulin [66, 67, 69]. Nous allons donc commencer par rappeler ce qui avait été fait à l'époque et préciser ensuite en quoi cela laissait la place à de nouveaux travaux dans ce domaine.

Il faut tout d'abord signaler que Coq est bâti directement sur l'isomorphisme de Curry-Howard : en particulier les preuves sont représentées en interne directement par des λ-termes. On est alors tenté de dire qu'une preuve Coq *est* précisément un programme fonctionnel, et que l'extraction n'a qu'une recopie à faire. Mais cette approche, bien que correcte, est trop naïve. En effet, si l'on reprend l'exemple d'une proposition existentielle $\exists$`x,P(x)`, une preuve intuitionniste de cette proposition contient bien la méthode de construction du témoin `x`, mais également des justifications logiques assurant que `x` convient bien, c'est-à-dire vérifie `P(x)`. Du point de vue programmation, le squelette constructif va nous donner le programme voulu, alors que les justifications logiques sont en règle générale non désirables dans le programme. Le rôle de l'extraction est alors de dériver à partir d'un terme `t` de type `T` un programme `p` ne renfermant plus que le contenu calculatoire de `t`. Et usuellement, la correction de ce programme `p` est garantie par une relation de *réalisabilité* `r` reliant `p` et le type `T` d'origine. Cette notion de réalisabilité a été introduite initialement par S. C. Kleene en 1945 dans [49]. La fonction d'extraction, que nous noterons $\mathcal{E}$, sera alors correcte si l'on peut établir qu'une relation de typage `t:T` implique la relation de réalisabilité $\mathcal{E}$`(t) r T`

après extraction.

Dans sa thèse, C. Paulin définit une notion de réalisabilité adaptée au Calcul des Constructions[3], et une extraction associée. Une première version de cette réalisabilité est basée sur un critère sémantique de « formules pré-réalisées ». Puis dans un deuxième temps, C. Paulin propose de remplacer ce critère sémantique par un critère syntaxique, à savoir l'annotation par l'utilisateur du caractère calculatoire ou au contraire logique des objets manipulés. Pour cela, au lieu d'utiliser un seul type `Prop` de toutes les propositions, on double ce type d'un type `Set` des propositions informatives[4], les propositions restant dans `Prop` étant considérées purement logiques.

Cette extraction théorique proposée et prouvée correcte par C. Paulin produit des programmes appartenant à une restriction du Calcul des Constructions, à savoir F_ω. Ces travaux ont été implantés dans l'assistant Coq par la suite, d'abord par C. Paulin pour la partie extraction vers F_ω, puis par B. Werner pour la partie allant de F_ω à ML[5] (voir [69]). Ultérieurement J.-C. Filliâtre a maintenu et amélioré cette implantation.

Les apports de la présente thèse

Ces travaux initiaux sur l'extraction Coq souffrent d'un certain nombre de limitations, certaines présentes dès la création, d'autres apparues au cours des évolutions du système Coq. Notre travail a essentiellement consisté à résoudre ces limitations, ce qui a conduit comme nous allons le voir à une refonte quasiment complète de ce mécanisme d'extraction. En même temps, nous nous sommes efforcés de maintenir le plus possible une compatibilité avec l'extraction précédente. Et cette nouvelle implantation de l'extraction a été progressivement intégrée dans les versions 7.0 et suivantes de Coq.

Prise en compte des univers

Une première limitation concerne les questions d'*univers* ou *sortes*. Nous avons déjà évoqué la division de `Prop`, l'univers des propositions, en deux univers, l'un `Set` pour les propositions calculatoires, l'autre, `Prop`, pour les propositions logiques. Mais ces deux univers `Prop` et `Set` font partie eux-mêmes d'un univers de niveau supérieur, `Type`. Or Coq traite ces univers comme tout terme du système. On peut ainsi former une quantification universelle sur `Type`, qui portera en particulier sur `Set` et `Prop`. L'extraction de C. Paulin n'était pas en mesure de travailler avec de tels termes incluant des raisonnements sur les univers. Tout terme faisant intervenir l'univers supérieur `Type` était tout simplement considéré comme non-extractible. Et il ne s'agissait pas là d'une restriction artificielle, mais bien d'une limitation intrinsèque de cette méthode d'extraction. Une étude aussi bien théorique que pratique de l'extraction au niveau `Type` restait donc à faire, ce qui s'est révélé être non-trivial. Cette limitation devenait d'autant plus gênante que l'usage de l'univers `Type` tend à se répandre dans les développements Coq. Par exemple, cet univers permet d'écrire des types de données

[3] Des Constructions pas encore Inductives à ce moment-là...

[4] Dans la thèse de C. Paulin, `Set` était nommée `Spec`

[5] Les différents langages « parlés » par l'extraction ont été Caml (Lourd puis `Light`), LazyML et maintenant Ocaml, Haskell et Scheme.

compatibles à la fois avec `Set` et avec `Prop`. D'autre part, `Type` est aussi fréquemment utilisé en association avec l'élimination forte dans des développements basés sur la réflexion (ou approche à deux niveaux, voir par exemple [16]). Ces développements étaient donc à l'origine hors du champ d'application de l'extraction Coq.

Résolution des problèmes de typage

Un second problème apparaît dans l'étape de traduction de F_ω vers un des langages fonctionnels concrets à la ML. Cette étape, présente dans l'implantation, n'est pas abordée par la thèse de C. Paulin. Or le système de types de F_ω est beaucoup plus riche que celui de ML. L'extraction peut tout à fait produire un terme extrait non typable en ML. En pratique, un tel conflit de typage se produit rarement, mais arrive tout de même. Et l'utilisation de plus en plus fréquente de la sorte `Type` tend à multiplier ces situations. De plus, cette faible fréquence des conflits de typage peut également s'expliquer par une forme d'auto-censure de la part des utilisateurs, peu enclins à utiliser `Type` s'ils savent à l'avance que cela rendra leur développement non-extractible. Nous avons mis au point une méthode consistant à identifier les emplacements de ces conflits de typage, puis à les résoudre en utilisant des fonctions bas-niveau influençant le typage. De la sorte, le code extrait est toujours utilisable avec le compilateur standard du langage cible. Pour l'instant cette méthode n'a été implantée que pour le langage Ocaml.

Correction de l'évaluation stricte

Le principal problème de l'ancienne extraction est sont manque de sécurité concernant l'exécution de termes extraits en suivant une stratégie stricte, comme en Ocaml. En fait, l'exécution de certains termes extraits en Coq version 5.x et 6.x peut terminer anormalement sur plusieurs erreurs fatales, ou bien au contraire boucler sans jamais terminer. Pour bien comprendre le problème, revenons tout d'abord à l'extraction d'origine vers F_ω. Dans sa thèse, C. Paulin a établi que son extraction produit des termes bien-typés dans F_ω. Ce système étant fortement normalisant, la réduction de ces termes extraits est donc assurée de se dérouler correctement, quelle que soit la stratégie employée, stricte ou paresseuse.

Mais ajoutons maintenant un axiome `A` dans Coq. Lorsqu'on extrait une preuve utilisant cet axiome, si l'on ne veut pas obtenir un programme extrait incomplet, il faut fournir manuellement un programme p correspondant à cet axiome `A`, c'est-à-dire le réalisant : `p r A`. Lorsque ce programme `p` est typable dans F_ω, le résultat précédent de correction tient encore, et toute réduction se déroulera sans problème. Malheureusement, trois axiomes particulièrement naturels et importants pour l'expressivité du système n'ont pas de réalisations typables dans F_ω. Tout programme extrait utilisant les réalisations de ces axiomes peut alors théoriquement voir son exécution échouer, et cela se produit effectivement dans certains cas, au moins lorsque la stratégie d'évaluation est stricte. Voici ces trois axiomes :

$$\frac{\bot}{A} \qquad \frac{x = y \quad P(x)}{P(y)} \qquad \frac{\mathcal{WF}(R) \quad \forall x, (\forall y, R(y,x) \to P(y)) \to P(y)}{\forall x, P(x)}$$

- Le premier axiome est l'élimination de l'absurde, qui permet de traiter les cas absurdes des preuves et programmes. Cet axiome correspond naturellement à une fonction levant une exception. Mais une telle réalisation conduit parfois à des évaluations terminant anormalement sur une exception non rattrapée.
- Le deuxième axiome est l'élimination de l'égalité. Avec la version la plus générale de cet axiome, une égalité entre types permet de modifier le typage apparent d'un terme. Cela ne pose pas de problèmes en Coq, par contre après extraction on peut avoir toutes sortes d'échecs d'exécution liés à des erreurs de typage, par exemple `(0 0)`.
- Le troisième axiome permet de prouver une propriété P par récurrence sur un prédicat R bien fondé. Cet axiome se réalise aisément par un opérateur de point-fixe à la `Y`. Mais en utilisant cette réalisation, l'ancienne extraction peut alors générer des termes dont l'évaluation stricte va boucler.

En fait, trouver des réalisations typables dans F_ω pour ces trois axiomes aurait conduit à considérer comme calculatoires l'absurde, l'égalité, et la bonne-fondation, soit à terme presque toutes les parties logiques. À défaut de cela, ces trois catégories de problèmes, dont deux sont déjà évoquées dans [69] ont été ignorées ou minimisées de façon empirique[6]. Après tout, si l'on souhaitait garantir la correction des termes extraits vers Ocaml, on pouvait toujours travailler dans une version bridée du système, sans ces trois axiomes, qui faisaient pourtant partie de la bibliothèque chargée initialement par le système.

Pour résoudre ces problèmes d'exécution, nous avons donc dû remanier la fonction $\mathcal{E}$ d'extraction en profondeur, et en particulier l'élimination des λ-abstractions logiques, afin de garantir une évaluation correcte quelle que soit la stratégie employée.

Support des évolutions du système

Depuis les premiers travaux sur l'extraction, l'évolution de Coq a accentué les limitations de cette ancienne extraction. Nous avons déjà mentionné par exemple le recours de plus en plus fréquent à la sorte `Type`. D'autre part, les trois axiomes qui mettent en péril la correction de l'exécution des termes extraits sont néanmoins essentiels pour l'expressivité du système. En particulier, lorsqu'il a été possible de modifier le système logique sous-jacent à Coq afin de le renforcer et de pouvoir *prouver* ces axiomes, cela a été fait. Et l'extraction s'est alors retrouvée confrontée aux situations incorrectes même sans le moindre ajout d'axiomes. Il devenait donc crucial de corriger ces problèmes, ce qui a été fait.

Parmi les autres évolutions de Coq ayant un impact sur l'extraction, on peut citer bien sûr le passage du Calcul des Constructions au Calcul des Constructions Inductives, c'est-à-dire l'ajout de types inductifs primitifs, ou encore beaucoup plus récemment l'adoption d'un système de modules et foncteurs. Parmi ces évolutions, certaines sont bénignes pour l'extraction. Par exemple, l'ajout de types enregistrements n'a fondamentalement rien changé, puisqu'ils ne sont visibles que par l'utilisateur et sont traduits en interne vers des types inductifs. De même, extraire des types co-inductifs ne demande aucun travail particulier si le langage cible est paresseux comme Haskell, et l'ancienne extraction gérait déjà ce cas.

[6] La fonction `False_rec`, levant l'exception associé à l'élimination de l'absurde, était ainsi toujours dépliée pour éviter la levée d'une exception dès sa définition.

Certains enrichissements nécessitent par contre des modifications plus substantielles de l'implantation, mais sans remise en cause de la théorie de l'extraction. C'est le cas par exemple de la gestion par B. Werner de l'extraction des types inductifs ou bien encore de notre adaptation de l'extraction au nouveau système de modules.

Modules et interfaces

Arrêtons-nous un instant sur cette extension aux nouveaux modules Coq de l'extraction vers Ocaml, et sur la génération d'une interface pour tout code extrait vers ce langage. Ces deux points peuvent sembler mineurs au regard des correctifs précédents de failles mettant en jeu l'exécution du code extrait. Mais il est essentiel à nos yeux que du code extrait puisse s'intégrer aisément dans un développement plus large. La génération des interfaces n'est en fait qu'une retombée positive de la correction des problèmes de typages évoqués précédemment, mais elle permet de prédire le type d'une fonction extraite uniquement à partir du type Coq de l'objet initial.

Bilan

Au final, notre nouvelle extraction se distingue donc par les trois points suivants :

1. Elle s'efforce de gérer n'importe quel terme Coq.
2. Elle assure que l'exécution des termes extraits va être correcte, aussi bien avec une stratégie paresseuse que stricte.
3. Elle garantit le bon typage du code extrait et fournit une interface à ce code (en Ocaml seulement pour l'instant).

On est donc passé en quinze ans d'une extraction encore expérimentale à un outil mature de développement de code certifié. En particulier cet outil est maintenant capable de traiter des exemples significatifs et réalistes, comme par exemple la bibliothèque d'ensembles finis utilisée en pratique par les développeurs Ocaml, basée sur des modules et des foncteurs (voir chapitre 7).

Comparaison avec d'autres systèmes d'extraction

Nous allons maintenant comparer l'extraction Coq avec les outils similaires présents dans d'autres systèmes. Cette comparaison va être centrée sur les deux points essentiels de l'extraction Coq, à savoir la suppression des parties logiques dans les preuves, puis la traduction vers un véritable langage de programmation fonctionnelle. Pour plus de détails, nous renvoyons au chapitre 6 de [66], qui reste largement d'actualité.

Suppression des parties logiques dans les preuves

Revenons tout d'abord sur la distinction entre partie logique et partie calculatoire d'un terme Coq. Dans le cas de l'exemple typique d'un terme de type $\forall$`x,P(x)`$\rightarrow\exists$`y,Q(x,y)`, nous avons vu que l'isomorphisme de Curry-Howard nous donne une fonction à deux arguments `x` et `p` et à deux résultats `y` et `q`. Si l'on veut obtenir au final une fonction produisant `y` à partir

de `x` seulement, il convient de s'assurer que les certificats logiques `p` et `q` n'interviennent pas dans cette construction de `y`. Si c'est bien le cas, `p` et `q` sont alors de simples décorations ou *code mort* du point de vue du calcul de `y`. Et normalement, la suppression de ce code mort amène des gains très importants concernant la taille du programme final et sa vitesse d'exécution.

Beaucoup de travail a été accompli dans le domaine de la détection automatique de code mort. À l'origine, S. Berardi a étudié cette élimination du code mort dans le contexte du λ-calcul simplement typé [12], avant que L. Boerio n'étende ces techniques au second ordre [15]. Puis F. Prost a généralisé ces travaux aux Pure Type Systems [72].

Nous avons cependant choisi de rester compatible avec l'ancienne extraction de Coq, en reposant toujours sur l'annotation des parties logiques par l'utilisateur. En pratique ceci n'est pas si contraignant, étant donné qu'il suffit d'annoter seulement chaque définition d'objet. Il faut donc simplement déterminer à l'avance, et une fois pour toute, le rôle de chaque objet. Au passage, ces annotations par les univers `Prop`, `Set` ou `Type` remplissent également un autre rôle que d'aider l'extraction (voir la question de l'imprédicativité dans le chapitre suivant).

Rappelons que `Prop` est essentiellement une copie de `Set`, mais avec une signification logique, alors que tout objet placé dans les univers `Set` et `Type` sera réputé informatif. Nous verrons par la suite que le système de type de Coq assure que les calculs sur des objets informatifs ne dépendent pas des résultats de calculs dans des parties logiques. Ceci empêche d'utiliser par mégarde une construction logique auxiliaire à un emplacement calculatoire, et justifie donc la suppression lors de l'extraction des objets ayant `Prop` comme univers.

En fait, cette méthode et l'analyse automatique de code mort sont relativement orthogonales : bien que l'extraction ne sache pas éliminer du code mort placé dans un univers informatif, elle peut à l'inverse simplifier des sous-termes qui ne satisfont pas les critères de code mort (voir en particulier l'élimination des inductifs singletons logiques en partie 2.3.2). Il ne serait d'ailleurs pas absurde de combiner les deux techniques, en soumettant par exemple le code extrait à une analyse de code mort. L'extraction, procédé syntaxique, permettrait alors de supprimer à peu de frais de larges pans logiques tout en laissant une analyse de code mort plus complexe et coûteuse travailler sur les termes ainsi dégrossis.

Si l'on étudie maintenant le système PX [44], on constate que l'élimination des parties logiques repose dans ce système sur un concept syntaxique de formules sans contenu calculatoire (dites de rang 0). On retrouve en particulier parmi ces formules à éliminer les formules de Harrop[7], mais également des annotations manuelles `◇A`, qui sont les formules `A` dont on a caché le contenu calculatoire.

En Nuprl [50], on trouve tout d'abord un certain nombre de formules prédéfinies comme étant de contenu nul. Il s'agit de l'égalité et de l'inégalité, l'absurde et la relation `a∈A` qui exprime que `a` est une preuve de `A`. À côté de cela, dans un type sous-ensemble `{x:A|B(x)}`, le rôle de `B` est assez similaire à celui d'un objet Coq dans `Prop` : on ne peut s'en servir que pour établir une formule de contenu nul ou bien une autre propriété à droite d'un type sous-ensemble.

[7] Il s'agit d'une classe bien connue de formules sans contenu calculatoire, regroupant les formules sans disjonction ou quantification existentielle en position positive.

Du côté de Minlog, on retrouve dans [30] la distinction entre formules de Harrop et autres formules réputées informatives. En outre, H. Schwichtenberg nous a confié lors d'une communication privée son intérêt pour l'utilisation de deux sortes de quantificateurs $\forall$ et $\forall_{nc}$, le second portant sur une variable sans contenu calculatoire. Signalons également au passage les nombreux travaux autour de Minlog dans le but d'extraire des programmes y compris à partir de preuves classiques.

Enfin, en Isabelle, S. Berghofer effectue également une analyse automatique de contenu calculatoire des termes à extraire, qui revient à éliminer les formules de Harrop. Par contre le statut des variables de prédicats n'est pas décidé tant que ces variables ne sont pas instanciées. Pour un théorème d'origine à n variables de prédicats, il peut être nécessaire de générer au pire 2^n variantes extraites pour gérer toutes les situations lors de l'utilisation ultérieure de ce théorème (voir p.64 de [14]). D'un autre côté, dans le cas de Isabelle/HOL, l'extraction repose sur une annotation manuelle pour ce qui est du contenu calculatoire ou non des prédicats inductifs (voir p.84 de [14]). Il faut noter qu'Isabelle est basée sur une logique classique. En fait, à la différence de Minlog, l'extraction se contente d'identifier les parties constructives des termes, sans chercher à travailler sur les parties classiques.

Traduction vers un vrai langage fonctionnel

Depuis le début, les versions successives de l'extraction en Coq ont toutes eu pour objectif de produire du code source pour des langages fonctionnels répandus, et non juste des λ-termes bruts. Actuellement, notre implantation accepte trois langages cibles : Ocaml [53], Haskell [31] et Scheme [48]. Parmi ces trois langages, l'extraction vers Ocaml est la plus complète et mature, alors qu'à l'inverse celle vers Scheme est encore expérimentale. Il y a trois raisons principales pour l'utilisation de tels langages cibles externes :

- Tout d'abord, le code certifié produit ainsi peut plus facilement s'intégrer dans des développements plus larges. Ceci est rendu possible par la production d'interfaces lisibles[8]. Ainsi, on peut par exemple obtenir un programme autonome en ajoutant à du code extrait une partie de gestion des entrées-sorties écrite à la main (voir les exemples du chapitre 5). De même, il est possible de développer une bibliothèque certifiée extraite, qui peut par la suite être réutilisée dans de multiples projets. Par exemple, le chapitre 7 présente la formalisation d'une bibliothèque d'ensembles finis. De la sorte, une large communauté de programmeurs peut bénéficier de l'extraction, y compris les non-utilisateurs de Coq.
- Ensuite, l'utilisation de tels langages externes permet des gains de vitesse substantiels. Comme il a été dit précédemment, une preuve Coq peut être vue directement comme un programme. Et son exécution directement dans Coq est possible via la $\beta\delta\iota\zeta$-réduction (cf. par exemple la commande `Eval Compute`). Mais lors de cette exécution, Coq se comporte essentiellement comme un interpréteur. Et il est bien connu que ceci est bien moins efficace que l'utilisation d'un compilateur. Une idée naturelle est alors d'utiliser des compilateurs déjà existants et reconnus, comme ceux pour Ocaml ou Haskell. Il est à noter cependant que la réduction interne dans Coq est en voie d'amélioration :

[8] En pratique, nous essayons même de produire du code source aussi lisible que possible.

B. Grégoire a travaillé à la réalisation d'un compilateur pour les termes Coq [39, 40]. Mais ce compilateur ne peut pas ignorer autant de parties logiques que l'extraction, car il doit être en mesure de travailler avec des termes non-clos et de réduire sous les lambdas. Et ceci, mêlé à l'élimination de certaines parties logiques, peut mener à la non-terminaison (voir partie 2.3.2).

- Enfin, une raison pragmatique de ce choix en faveur de langages cibles externes est l'impossibilité d'écrire dans Coq certains termes extraits. On a vu qu'à l'origine, les travaux de C. Paulin sur l'extraction en Coq passaient par une étape intermédiaire d'extraction interne. On trouve également cette idée d'extraction interne chez P. Severi & N. Szasz [76]. Mais le système théorique actuellement utilisé par Coq diffère sensiblement de ceux utilisés dans ces études. Et les mêmes enrichissements que nous avons vus remettre en cause la correction de l'ancienne extraction, empêchent également de réaliser dorénavant une extraction interne complète. En particulier, Coq accepte maintenant la définition d'un point-fixe informatif basé sur une mesure de décroissance logique. C'est le cas par exemple de la relation d'accessibilité `Acc` et de ses points-fixes associés `Acc_rec` et `Acc_iter` que nous étudierons par la suite. Une extraction interne de ces termes conduirait alors à un point-fixe sans mesure de décroissance, et donc potentiellement à des termes non-fortement normalisants (voir un exemple dans la partie 2.3.2). Or de tels objets à réduction infinie sont proscrits en Coq.

A contrario, il est évident que l'utilisation d'un langage cible interne rend beaucoup plus simple la justification de la correction de l'extraction. En effet, à partir d'une preuve (pas forcément formelle) de correction par réalisabilité d'une extraction interne, on peut assez facilement implanter une procédure qui construit explicitement, en plus d'un terme extrait, la preuve que ce terme extrait particulier réalise bien sa spécification. De tels outils de génération automatique de preuves particulières de correction ont ainsi été implantés en Minlog et Isabelle. Le cas d'une extraction externe n'exclut évidemment pas d'obtenir une preuve formelle de correction. Mais il faut alors ajouter une étape supplémentaire au processus, à savoir une formalisation de la sémantique de notre langage cible, afin de pouvoir exprimer l'adéquation entre le terme extrait, cette sémantique et la spécification initiale. Faute de temps, cette preuve de correction n'a été faite au cours de nos travaux que sur le papier, et fait l'objet du chapitre 2.

Il faut noter que ces deux systèmes ne sont pas confrontés aux problèmes venant de la richesse de la logique de Coq. En particulier, les fonctions extraites en Minlog sont typables dans le système T de Gödel, et on ne peut obtenir par extraction les récurrences non structurelles (mais bien-fondées) qui rendent impossible, entre autres raisons, une extraction interne en Coq. À chacun de faire la part des avantages et inconvénients entre d'un côté une logique riche et une extraction complexe mais expressive, et de l'autre une logique plus minimale et une extraction simple.

En outre, S. Berghofer mentionne également dans [14] la possibilité de générer du véritable code ML à partir de ses termes extraits internes. Cette étape, même si elle semble triviale compte tenu de la proximité entre ses termes internes et la syntaxe ML, doit néanmoins être faite avec la plus grande prudence. On peut en effet faire un parallèle avec la

génération de termes internes F_ω dans l'ancienne extraction Coq, et leur traduction ultérieure vers ML en une étape informelle. Or nous venons de voir que cela pouvait aboutir à des termes extraits dont l'évaluation stricte échoue, malgré un résultat de correction au niveau F_ω.

Quant à Nuprl, l'extraction y produit également un λ-terme interne, que l'on peut réduire en utilisant une machine à réduction interne à Nuprl. Mais il ne semble pas actuellement que la preuve de correction d'un λ-terme extrait puisse être obtenue automatiquement.

Plan

Ce travail débute par une introduction progressive à l'assistant de preuve Coq. La première partie du chapitre 1 présente via des exemples les principales caractéristiques de ce système, et en particulier celles qui vont avoir une influence sur l'extraction. La deuxième partie de ce chapitre expose plus formellement Cci, le système théorique sous-jacent.

Après ce bref tour d'horizon de Coq et du Cci, nous abordons une première moitié du manuscrit dédiée à la présentation de notre nouvelle extraction. Le chapitre 2 présente tout d'abord notre refonte de la fonction d'extraction $\mathcal{E}$ sur les termes, ainsi que deux preuves complémentaires de correction. La première preuve est une preuve syntaxique de bon déroulement de la réduction d'un terme extrait, la seconde preuve est une preuve sémantique de correction des termes extraits vis-à-vis de leurs spécifications.

Le chapitre 3 est ensuite dédié au typage des termes extraits. Nous commençons par identifier les problèmes liés au (non-)typage des termes extraits dans un système de type à la ML, puis nous proposons une solution de contournement de ces difficultés.

Enfin, le chapitre 4 complète la description de notre nouvelle extraction en présentant les derniers aspects de notre travail : extraction des modules, des types co-inductifs, des types enregistrements, et enfin optimisation du code extrait. Enfin nous présentons succinctement l'implantation réalisée, du point de vue développeur puis utilisateur.

Le chapitre 5 entame la seconde moitié du manuscrit, consacrée à des études de cas. Dans ce chapitre, nous commençons par un tour d'horizon des contributions utilisateurs du point de vue de l'extraction, ces contributions utilisateurs formant une bibliothèque déjà conséquente d'exemples de développements Coq. Puis nous détaillons le cas de deux de ces contributions, qui mettent en défaut l'ancienne extraction : la première sur les exceptions par continuations, due à J.-F. Monin, et la seconde sur le lemme de Higman, par H. Herbelin.

Le chapitre 6 fait le point sur un projet ambitieux consistant à développer une bibliothèque d'arithmétique réelle exacte par extraction d'un développement d'analyse réelle constructive. Le développement en question est le projet C-CoRN de l'université de Nimègue. Nous verrons que l'on est encore loin du but, mais qu'en même temps d'énormes progrès ont déjà été accomplis. D'autre part, nous présentons une petite expérimentation alternative autour des réels constructifs faite en collaboration avec H. Schwichtenberg.

Enfin, le chapitre 7 présente la certification de la bibliothèque Ocaml d'ensembles finis que nous avons effectuée en collaboration avec J.-C. Filliâtre. Ce développement est l'un des premiers à combiner modules, foncteurs et extraction.

Conventions syntaxiques

Tout au long de ce document, les fragments de code source donnés en exemple seront mis en évidence ainsi :

```
Definition zero := 0.
```

Le contenu de ces fragments pourra être en Coq, Ocaml ou Haskell selon le cas. Les exemples courts seront mentionnés avec seulement un changement de police, comme par exemple `(plus 0 0)`.

On rencontrera également quelques exemples d'interactions de l'utilisateur avec Coq :

```
Coq < Eval compute in 1+1.
    = 2
    : nat
```

Nous reprenons ici les usages de l'ancienne interface texte `coqtop` : les lignes précédées de l'invite « `Coq <` » correspondent aux saisies de l'utilisateur, les autres lignes étant la réponse de Coq. De même, les exemples d'utilisation de la boucle interactive de Ocaml seront présentés comme suit, avec le caractère « `#` » comme invite :

```
# 1+1;;
- : int = 2
```

Les exemples Coq sont destinés à être utilisés avec Coq version 8.0 ou ultérieure [78]. Il est à noter que cette version inaugure une nouvelle syntaxe, beaucoup plus agréable. Au delà de l'usage de cette nouvelle syntaxe, nous avons également amélioré la lisibilité des exemples en tirant parti de certaines fonctionnalités récentes du système :

- Certains mots-clés ASCII d'origine ont été remplacés par leurs équivalents Unicode[9] :

forall	exists	->	<->	<>	~	/\	\/	<=
∀	∃	→	↔	≠	¬	∧	∨	≤

- Nous avons utilisé la syntaxe la plus lisible pour les expressions arithmétiques, par exemple `0+1` au lieu de `(plus 0 (S 0))`, comme le permet le mécanisme de `Scope`.
- Quand le type d'une quantification peut être déduit du contexte, Coq autorise maintenant son omission. Nous utilisons parfois cette possibilité.

[9] Ceci peut réellement se faire en Coq, via la commande `Notation` et l'utilisation d'une interface compatible avec l'Unicode, comme par exemple CoqIDE [78].

Chapitre 1

Une présentation de Coq

Ce chapitre a un double objectif :

- Tout d'abord le lecteur découvrant l'assistant de preuve Coq trouvera ici une description rapide de ce système, et plus particulièrement du système logique sous-jacent, à savoir le Calcul des Constructions Inductives (Cci en abrégé). Idéalement, il n'est besoin ici que de connaissances de base en théorie des types et en λ-calcul. Bien sûr, ce chapitre n'a pas pour vocation de remplacer la documentation accompagnant le système. Le lecteur pourra donc consulter en complément :
  - le Tutoriel [47] pour une introduction plus graduelle,
  - le Manuel de Référence [78], et en particulier son chapitre 4 sur le Cci, pour une description formelle exhaustive de Coq.
- En même temps, cette présentation de Coq est bien sûr axée sur l'extraction. Même si cette extraction ne sera abordée qu'au chapitre suivant, nous allons détailler ici toutes les caractéristiques et particularités du système logique Cci qui vont avoir des répercussions au niveau de l'extraction.

1.1 Une introduction par l'exemple

1.1.1 Coq comme système logique

L'objectif premier de Coq est d'être un assistant de preuve, et donc de permettre de formaliser le raisonnement mathématique. Prenons un énoncé trivialement vrai : $A \to A$, où A est une proposition quelconque. En voici une preuve en déduction naturelle (cf. [71]) :

$$\dfrac{\dfrac{}{A \vdash A}(Ax)}{\vdash A \to A}(\to I)$$

Cette preuve peut être directement transcrite dans le système Coq. On se donne tout d'abord une variable propositionnelle.

```
Parameter A : Prop.
```

Nous reviendrons plus tard sur ce `Prop`, mais ici on peut le voir simplement comme l'ensemble des propositions logiques.

```
Lemma easy : A → A.
Proof.
 intro.
 assumption.
Qed.
```

Cette portion de script Coq commence par le nom et l'énoncé de notre lemme. Puis entre les mots-clés `Proof` et `Qed` se trouve la preuve de cet énoncé, constituée de directives ou *tactiques*. Ces tactiques reflètent ici la structure de la preuve en déduction naturelle. La tactique `intro` correspond à l'utilisation de la règle d'introduction de la flèche ($\rightarrow I$). Quant à la tactique `assumption`, elle correspond à la règle axiome (Ax).

Le script présenté ici est complet, mais le système Coq, par sa nature *interactive*, permet de bâtir cette preuve par étapes. Ainsi, si l'on s'arrête après la commande `intro`, le système affiche l'état courant du ou des buts à prouver :

```
1 subgoal

  H : A
  ============================
   A
```

Il s'agit alors de trouver une preuve de `A` sous l'hypothèse `A` (cette hypothèse étant nommée `H`). De nombreuses interfaces existent pour faciliter cette interaction avec le système Coq, comme par exemple Proof-General [4], Pcoq [3], ou plus récemment CoqIDE [78]. L'extraction travaillant sur des preuves terminées, nous ne développerons pas cette notion d'interaction.

Il faut également noter que Coq met à la disposition de l'utilisateur une grande variété de tactiques. Ici par exemple, pour un énoncé aussi simple, la tactique de recherche automatique `auto` aurait suffit à bâtir directement la preuve. Là encore, nous ne détaillerons pas ces tactiques. En effet, l'extraction ne se préoccupe pas du cheminement suivi pour obtenir la preuve, mais de la forme finale de la preuve. En l'occurrence, `auto` bâtit ici la même preuve en interne que `intro` suivi de `assumption`.

Des preuves sous forme de λ-termes

Quelle est donc cette représentation interne des preuves ? Il s'agit de λ-termes, en application de l'isomorphisme de Curry-Howard. Le système logique sous-jacent à Coq, le Cci est en effet un λ-calcul avec un système de types puissant. Et toujours en application de l'isomorphisme de Curry-Howard [46, 7, 38], les énoncés exprimables en Coq sont des types du Cci. Enfin, vérifier si `t` est bien une preuve valide de l'énoncé `T` consiste à vérifier que le type `T` est bien un type légal du λ-terme `t`.

Demandons à Coq quelle est la représentation interne du lemme `easy` :

```
Coq < Print easy.
```

.../...

.../...

```
easy = fun H : A ⇒ H
     : A → A
```

La syntaxe `fun x:X ⇒ t` est la notation Coq pour la λ-abstraction typée λ`x:X.t`. Cette abstraction est ici l'effet de la tactique `intro`. Quant à `assumption`, son effet est l'utilisation d'une variable du contexte, ici `H`. De façon plus générale, toute tactique contribue à construire petit à petit le λ-terme de la preuve. Et au final, il est clair que `fun H:A ⇒ H` est bien de type `A→A`.

Ces preuves sous forme de λ-termes sont habituellement maniées uniquement par le système. Mais rien n'empêche l'utilisateur de fournir tout ou partie de sa preuve sous cette forme, par exemple via la tactique `exact`.

```
Lemma easy : A → A.
Proof.
 exact (fun a ⇒ a).
Qed.
```

On remarquera ici que Coq est en mesure d'inférer le type de `a`, qui est nécessairement A. Il est alors facultatif d'écrire ce type. Nous utiliserons fréquemment cette possibilité par la suite.

Enfin si le λ-terme complet est connu à l'avance, on peut le donner directement sous forme d'une `Definition`, ce qui donne ici :

```
Definition easy : A → A := fun a ⇒ a.
```

Alternativement, on peut également utiliser un style de définition où les arguments sont nommés dès la déclaration du type, rendant inutile l'écriture des λ-abstractions de tête :

```
Definition easy (a:A) : A := a.
```

Ce style est certes plus concis, mais pas forcément plus lisible. Nous l'éviterons donc, sauf dans le cas des fonctions récursives où il est indispensable (cf. plus bas).

L'ordre supérieur

Jusqu'ici, le seul type de base `A` rencontré avait été fixé comme paramètre, et l'énoncé du lemme `easy` portait sur ce `A` particulier. Mais on peut également considérer en Coq des énoncés (c'est-à-dire des types) portant sur des ensembles de types. Par exemple `∀A:Prop, A→A` est un énoncé (*i.e.* un type Coq) qui se lit : « pour toute proposition `A` (c'est-à-dire objet `A` appartenant au type `Prop`), `A` implique `A` ». Une telle quantification universelle typée `∀x:X, T` est également nommée *produit* en Coq. Cette quantification n'étant pas restreinte, le Cci est donc une logique d'ordre supérieur. On parle également de *types dépendants*, puisque le plus souvent le corps `T` du produit dépend de la variable `x` de la tête du produit.

Quelle va être la preuve de l'énoncé `∀A:Prop, A→A` ? L'introduction du « pour tout » consiste à se donner un objet quelconque dans le domaine, puis à mener le reste de la preuve avec cet objet. Au niveau du λ-calcul, ceci se traduit par une λ-abstraction : en réponse à un énoncé `∀A:Prop, ...`, la preuve va donc débuter par `fun A:Prop ⇒ ...`. La suite de la

preuve est alors la même que celle de notre lemme `easy`, ce qui donne finalement le λ-terme complet `fun A:Prop ⇒ fun a:A ⇒ a`, ou plus simplement `fun (A:Prop)(a:A) ⇒ a`.

Il est à remarquer que les λ-abstractions servent à la fois à bâtir les preuves de produits et de types flèches. Ceci n'est pas un hasard, on peut en effet voir un type flèche `A→B` comme un produit non-dépendant `∀x:A, B` pour lequel `x` n'apparaît pas dans `B`. En fait, en Coq, la syntaxe `A→B` est directement du sucre syntaxique pour `∀_:A, B`.

1.1.2 Coq comme langage de programmation

On peut également aborder Coq depuis l'autre versant de l'isomorphisme de Curry-Howard, et regarder le Cci non pas comme un système logique, mais plus comme un λ-calcul, c'est-à-dire un langage de programmation. Par exemple avec cette vision, notre lemme `easy` est la fonction identité sur le type `A`.

Des types de données inductifs

Pour que cette vision de Coq comme langage de programmation soit fructueuse, il nous faut pouvoir définir les types de données que l'on rencontre habituellement dans les autres langages de programmation. La définition de ces types de données peut se faire en Coq de façon aisée via l'usage des types inductifs. Tous les types inductifs que nous allons présenter dans ce chapitre font partie de la bibliothèque standard de Coq.

Le type des booléens s'obtient par la déclaration :

```
Inductive bool : Set := true : bool | false : bool.
```

Cette déclaration crée un nouveau type, nommé `bool`, ainsi que deux constructeurs `true` et `false` de type `bool`. L'annotation `Set` permet de désigner quel va être le type du type `bool`. Nous reviendrons par la suite sur ce point ; en attendant `Set` peut être vu comme l'ensemble des types de données. De même, on définit ainsi les entiers de Peano :

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

Le constructeur `O` code le zéro, et `S` code la fonction successeur. Un dernier exemple usuel est celui des listes paramétriques :

```
Inductive list (A:Set) : Set :=
 | nil : list A
 | cons : A → list A → list A.
```

Ici, la syntaxe `(A:Set)` exprime le fait que `list` dépend d'un paramètre `A`. Et lors de l'utilisation de cette structure de données, le paramètre sera fourni en argument, une liste d'entiers étant par exemple `list nat`.

Il faut noter que le système refuse certains inductifs dont la définition est pourtant syntaxiquement valide. Ces restrictions, qui ont pour but d'assurer la cohérence du système du point de vue logique, sont constituées essentiellement d'une condition de positivité. Le lecteur intéressé par plus de détails pourra consulter le chapitre 4 de [78] ou bien encore [68].

Le filtrage sur les termes inductifs

Pour tirer partie au mieux de ces types inductifs, le Cci est équipé d'un opérateur primitif permettant le filtrage, dont la syntaxe est `match ... with ...` . Ceci permet par exemple de définir le prédécesseur d'un entier.

```
Definition pred : nat → nat :=
 fun n ⇒
  match n with
   | O ⇒ O
   | S m ⇒ m
  end.
```

Le filtrage primitif de Coq est un filtrage simple, de premier niveau :

- À chaque branche correspond un constructeur et un seul.
- Dans la branche correspondant au constructeur `C`, le motif reconnu est forcément l'application de `C` à des variables.

Outre ce filtrage primitif, on a désormais en Coq la possibilité de définir des filtrages plus complexes, comme dans ce double prédécesseur :

```
Definition predpred : nat → nat :=
 fun n ⇒
  match n with
   | S (S m) ⇒ m
   | _ ⇒ O
  end.
```

Nous ne détaillerons pas ces filtrages complexes, qui sont en fait traduits au niveau interne de Coq vers une succession de filtrages primitifs. Notre double prédécesseur est ainsi l'imbrication de deux prédécesseurs simples.

Enfin, signalons pour être complet que la syntaxe `match ... with` accepte des annotations supplémentaires via les mot-clés `as`, `in` et `return`. Ces annotations sont presque toujours facultatives. Mais dans certaines situations Coq ne sait pas inférer de lui-même un type de sortie de filtrage, ou bien le fait mal. Ces annotations additionnelles permettent alors de fournir explicitement ce type de sortie. Nous rencontrerons ce cas ultérieurement, par exemple dans le terme `Acc_inv` (voir section 1.1.4).

La récursion structurelle

La dernière construction primitive du Cci est la possibilité de définir un terme par récurrence structurelle sur un objet inductif. Dans sa version la plus simple, cette récursion structurelle correspond à la syntaxe `Fixpoint`. Définissons par exemple l'addition de deux entiers unaires :

```
Fixpoint plus (n m:nat) {struct n} : nat :=
```

.../...

```
...
 match n with
  | O ⇒ m
  | S n' ⇒ S (plus n' m)
 end.
```

Le type complet `nat→nat→nat` de `plus` est ici scindé entre deux arguments nommés `n` et `m` d'un côté du « : » et un résultat `nat` de l'autre. Ceci permet de préciser via le `struct` sur quel argument inductif va porter la récursion. On parle alors d'argument inductif « de garde » ou « de décroissance ». Un autre choix possible de présentation aurait été de n'écarter que le premier argument, qui est alors implicitement l'argument « de garde », et de laisser comme type du résultat `nat→nat` :

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

Une définition récursive n'est acceptée que si tout appel récursif interne se fait sur un argument récursif qui est structurellement plus petit que l'argument récursif initial. Ici par exemple `n'` est bien un sous-terme de `n`. On trouvera une définition plus précise de ce « structurellement plus petit » dans [78]. La préoccupation est, là encore, celle de la cohérence logique. Sans ces conditions sur les appels récursifs, il serait en effet aisé de bâtir des termes de n'importe quel type `A` :

```
Fixpoint loop (n:nat) : A := loop n.

Definition impossible : A := loop 0.
```

Le système logique serait alors incohérent.

En fait, la syntaxe `Fixpoint` n'est pas la plus générale, car il s'agit d'une déclaration nommant immédiatement une fonction, et non un terme anonyme. En particulier on ne peut pas imbriquer un `Fixpoint` dans un autre. Un tel point-fixe anonyme peut se définir en Coq par la syntaxe `fix`. Voici une définition alternative de `plus` à l'aide de cette syntaxe :

```
Definition plus :=
 fix plusrec (n m:nat) {struct n} : nat :=
  match n with
   | O ⇒ m
   | S n' ⇒ S (plusrec n' m)
  end.
```

Au niveau interne de Coq, ces deux définitions de `plus` sont identiques. En fait, `Fixpoint` n'est que du sucre syntaxique pour une `Definition` suivi d'un `fix`, comme peut le montrer un `Print plus`. Un exemple typique d'usage d'un `fix` anonyme est la fusion de deux listes triées, qu'on trouvera par exemple p.195 dans une des implantations d'ensembles finis.

La réduction sur les termes

Le principal intérêt d'un langage de programmation est de pouvoir exécuter les programmes. Ceci est possible avec Coq. En effet le système logique Cci est équipé de règles de réductions :

(*beta*) Cci étant à la base un λ-calcul, on retrouve évidemment la β-réduction

(*delta*) Comme ce système autorise l'ajout de nouvelles constantes (via `Definition`, `Lemma`, etc), une règle de réduction nommée δ permet de remplacer un nom de constante par le corps de la constante.

(*zeta*) Coq comporte maintenant des « let in » primitifs, c'est-à-dire des abréviations locales `let x:=t in u`. Une règle du système nommée ζ permet de déplier ces abréviations, en remplaçant `x` par `t` partout dans `u`.

(*iota*) Le système comporte également deux règles dédiées aux inductifs, toutes les deux nommées ι. La première règle permet de réduire un filtrage, si le terme filtré commence par un constructeur. Par exemple, `match S n with O ⇒ O | S m ⇒ m end` peut se réduire en `n`. L'autre règle liée aux inductifs est celle qui permet de réduire une fonction récursive, dès que son argument récursif est présent et qu'il commence par un constructeur.

Ces réductions peuvent être utilisées à tout moment en Coq, par exemple via la commande `Eval compute`[1] :

```
Coq < Eval compute in pred (plus 2 2).
    = 3
     : nat
```

Ces calculs en Coq possèdent des propriétés très fortes par rapport à des exécutions dans des langages de programmation plus usuels :

(i) Tout d'abord, la réduction Coq est une réduction *forte*, c'est-à-dire possible à tout endroit, même sous un lambda ou dans le corps d'une fonction récursive. Par exemple :

```
Coq < Eval compute in fun n ⇒ plus 1 n.
     = fun n : nat ⇒ S n
     : nat→nat
```

Par comparaison, la plupart des langages usuels gèlent le corps des fonctions, pour ne les réduire que lorsque tous les arguments attendus sont présents.

(ii) Ensuite, toute réduction Coq est finie. Cette propriété est nommée normalisation forte. Pour un terme de départ donné, toute suite de réductions aboutit donc en un nombre fini d'étapes à une *forme normale*, c'est-à-dire un terme non réductible. Ceci résulte des multiples contraintes exigées avant qu'un terme soit accepté comme valide dans le Cci, ainsi que des contraintes sur les ι-réductions. À l'opposé, dans l'immense majorité des langages, il est très simple d'écrire un programme ne terminant pas.

[1] La syntaxe `2` est par défaut traduite en `S (S O)`, et ainsi de suite pour les autres nombres. Le nouveau mécanisme de `Scope` [78] permet de changer cette traduction selon les besoins ($\mathbb{Z}$, $\mathbb{R}$, ...).

(iii) D'autre part Coq vérifie également la propriété de confluence : si l'on considère deux réduits d'un même terme, il existe deux dérivations de ces deux termes vers un réduit commun. Ceci, associé avec (ii), permet de montrer que pour chaque terme initial il existe en fait une et une seule forme normale. On est donc sûr d'y aboutir en temps fini, peu importe l'ordre dans lequel on mène les calculs. Cette propriété n'est par exemple pas vérifiée par les langages ayant des effets de bord, pour lesquels l'ordre d'évaluation peut influer sur le résultat.

La contrepartie de ces propriétés méta-théoriques fortes est un certain manque d'expressivité de Cci vu comme langage de programmation. Impossible en effet d'y parler directement de fonctions partielles, c'est-à-dire non définies partout. Impossible également de définir directement une fonction récursive quelconque, non structurelle. En fait nous allons voir plus tard comment contourner ces deux limitations, via l'utilisation de pré-conditions pour le premier point et d'opérateurs de récursion bien-fondée dans le deuxième.

1.1.3 Les sortes de Cci

Nous allons maintenant aborder les arcanes de Coq au cœur du mécanisme d'extraction, à savoir les *univers* ou *sortes* de Coq. Il faut savoir qu'il n'y a pas en Coq de distinction syntaxique entre les termes de base (comme `0`) et les types (comme `nat`). Dans les deux cas, il s'agit de termes du Cci. Or tout terme du Cci possède un type, qui est de nouveau un terme du Cci. On peut donc se demander quel est le type du type, ou encore le type du type du type. Demandons au système :

```
Coq < Check 0.
0
     : nat
Coq < Check nat.
nat
     : Set
Coq < Check Set.
Set
     : Type
Coq < Check Type.
Type
     : Type
```

On voit apparaître `Set` et `Type`, qui avec `Prop` sont les trois objets spéciaux de Coq nommés *sortes*. Et ces sortes vont avoir un lien étroit avec les types de types du système :

- Tout d'abord, il est clair que `Set` et `Type` sont des types de types (de `0` et `nat` respectivement). Quant à `Prop`, il suffit de définir un inductif dans `Prop`, ce qui se fait d'une façon similaire à la définition (déjà vue) d'inductifs dans `Set` :

```
Inductive True : Prop := I : True.
```

On obtient alors que `Prop` est le type du type du constructeur `I`.

- D'autre part, une propriété du système Cci affirme que tout type de type peut se réduire vers `Set`, `Prop` ou `Type`.

Enfin, pour être complet, notons que le type de `Prop` est `Type`.

La sorte `Type`

Examinons tout d'abord la sorte `Type`. Il s'agit d'un objet étrange qui semble être son propre type. Si l'on prend une vision ensembliste des types, cela correspond à un ensemble d'ensembles se contenant lui-même, ce qui mène au paradoxe de Russel. Même si cette vision ensembliste est imparfaite, il est néanmoins vrai que `Type:Type` permet de prouver l'incohérence du système [20]. Un remède est alors de prendre une hiérarchie infinie de sortes. En Coq, les sortes `Type` sont implicitement indicées par un entier, et pour tout indice i on a : $\texttt{Type}_i : \texttt{Type}_{i+1}$. Du point de vue l'extraction, nous n'aurons pas besoin de distinguer parmi cette infinité de sortes, et nous resterons donc au niveau de l'approximation `Type:Type`.

La classification des sortes

On peut alors se demander la raison de la présence de trois sortes quand une seule, `Type`, aurait pu suffire. Deux critères indépendants permettent en fait de distinguer ces trois sortes :

- le caractère prédicatif ou bien imprédicatif
- le caractère logique ou bien calculatoire (nous dirons également informatif).

Nous allons maintenant détailler ces deux critères successivement. Mais avant cela, voici comment se placent les sortes de Coq vis-à-vis de ces deux critères. Jusqu'à la version 7.4 de Coq, la situation pouvait se résumer par le schéma suivant :

	imprédicatif	prédicatif
calculatoire	`Set`	`Type`
logique	`Prop`	

À compter de la version 8.0, la sorte `Set` est maintenant par défaut prédicative, ce qui nous donne le nouveau schéma suivant :

	imprédicatif	prédicatif
calculatoire		`Set`,`Type`
logique	`Prop`	

L'imprédicativité

Même si le caractère (im)prédicatif n'influe par sur l'extraction, nous allons quand même tenter d'illustrer cette notion. Pour une sorte imprédicative comme `Prop`, on autorise la

création d'un élément de `Prop` via une quantification universelle sur tous les éléments de `Prop` (et donc en particulier sur ce nouvel élément en instance de création). Par exemple, on a déjà rencontré le type de l'identité sur `Prop` :

```
Definition typeId : Prop := ∀A:Prop, A → A.
```

Par contre, il n'est pas possible de définir le même terme avec la sorte prédicative `Type` à la place de `Prop`. Cela marche apparemment, mais il s'agit d'une apparence seulement, car que les indices implicites des deux occurrences de `Type` seront en fait différentes :

```
Definition typeId' : Type_{i+1} := ∀A:Type_i, A → A.
```

Le problème est encore plus flagrant avec `Set` et Coq 8.0. La définition suivante :

```
Definition typeId'' : Set := ∀A:Set, A → A.
```

engendre une erreur expliquant que `typeId''` est de type `Type` et non `Set`.

Comme mentionné précédemment, la version 8.0 de Coq a vu la sorte `Set` devenir prédicative par défaut. En fait, une option de la ligne de commande de Coq 8.0 permet de revenir à la situation précédente. Dans tout ce qui va suivre, nous allons nous placer dans la situation par défaut de Coq 8.0, à savoir avec `Set` prédicatif, sauf mention explicite du contraire (voir par exemple l'étude du lemme de Higman en section 5.4). De toute façon, l'étude théorique de l'extraction qui va suivre est indépendante de cette question d'imprédicativité. Nous continuerons donc par abus de parler de CCI pour l'un ou l'autre système logique obtenu avec ou sans l'imprédicativité de `Set`, même si le nom officiel du système sans imprédicativité est désormais PCIC (pour « Predicative Calculus of Inductive Constructions »).

Sortes logiques et sortes calculatoires

Maintenant l'autre distinction entre les sortes concerne leur caractère logique ou bien calculatoire. Il s'agit au départ d'une simple convention d'usage entre `Prop` et `Set` :

- `Set` est destiné à contenir tous les objets ayant un contenu calculatoire, en particulier les exemples vu précédemment de types de données usuels pour le programmeur.
- `Prop`, à l'inverse, a pour rôle de contenir tout ce qui a trait à la logique pure, comme par exemple les justifications diverses, pré- et post-conditions, autrement dit tout ce qui peut être ignoré pendant des calculs.

L'utilisateur choisit donc lors de la définition d'un objet de le placer dans `Prop` ou dans une autre sorte. Et ce choix va être repris par l'extraction, qui va ignorer les parties logiques placées dans `Prop`.

Au niveau de la dualité logique/calculatoire, `Type` joue un rôle ambigu. En effet, CCI contient un principe de *cumulativité*, qui permet d'affirmer en particulier que si `t:Set`, alors on a également `t:Type`, et de même si `t:Prop`, alors on a également `t:Type`. Considérons l'identité sur `Type` :

```
Definition id : ∀X:Type, X → X := fun (X:Type) (x:X) ⇒ x.
```

La cumulativité fait que les termes suivants sont bien typées : `(id Set nat)` ou `(id nat O)` ou encore `(id Prop True)` ou enfin `(id True I)`. Par la cumulativité, certains termes de sorte `Type` sont donc en fait logiques car à l'origine dans `Prop`, et d'autres sont calculatoires. Dans l'incertitude, on doit alors considérer que les objets dans `Type` peuvent avoir un contenu calculatoire. Ceci fait qu'au niveau de l'extraction, `Set` et `Type` ne vont pas être distingués.

Les inductifs logiques

Concernant les opérations logiques, nous n'avons présenté jusqu'à maintenant que l'implication et la quantification universelle. Voyons maintenant comment définir les autres opérateurs logiques usuels via des inductifs dans `Prop`. On a déjà rencontré `True`, avec son constructeur unique `I`. Au niveau logique l'énoncé `True` admet donc une preuve immédiate, qui est simplement `I`. À l'opposé, `False` est un type inductif sans constructeur, et n'est donc pas prouvable dans un contexte vide (le contraire aurait pour conséquence l'incohérence du système).

```
Inductive False : Prop := .
```

La négation ¬`A` est alors définie comme `A`→`False`.

Les connecteurs logiques `or` et `and` sont définis comme suit :

```
Inductive or (A B:Prop) : Prop :=
 | or_introl : A → or A B
 | or_intror : B → or A B.

Inductive and (A B:Prop) : Prop :=  conj : A → B → and A B.
```

On rencontrera ces deux inductifs avec les syntaxes Coq ∧ et ∨.

Voici maintenant la quantification existentielle :

```
Inductive ex (A:Type)(P:A→Prop) : Prop :=
   ex_intro : ∀x:A, P x → ex A P.
```

La syntaxe Coq pour cette existentielle est ∃`x:A, P` (ou bien ∃`x, P` si `A` est inférable).

Enfin l'égalité correspond à un inductif ayant un seul constructeur mimant la réflexivité :

```
Inductive eq (A:Type)(x:A) : A → Prop :=  refl_equal : eq A x x.
```

L'égalité sera notée `=`, et la différence (c'est-à-dire la négation de l'égalité) sera noté ≠ .

Grâce au filtrage sur ces inductifs, on peut prouver les propriétés usuelles (intuitionnistes) de tous ces opérateurs logiques. Voici un exemple :

```
Definition proj1 : ∀A B:Prop, A∧B → A :=
 fun (A B:Prop)(ab:A∧B) ⇒ match ab with (conj a b) ⇒ a end.
```

Les règles d'élimination des inductifs

En fait, la dichotomie entre `Prop` utilisé comme sorte logique et `Set` utilisé comme sorte calculatoire est plus qu'une affaire de convention d'utilisation. En effet les règles autorisant ou non l'élimination d'un terme inductif (c'est-à-dire un filtrage sur ce terme) diffèrent selon la sorte de ce terme inductif. S'il s'agit de `Prop`, alors cette élimination ne peut servir qu'à construire un terme dans `Prop`. Par contre, on autorise les éliminations d'inductifs sur `Set` et `Type` à bâtir des termes de toutes les sortes[2]. L'idée est qu'un inductif logique, donc sans contenu calculatoire, ne doit pas influencer un calcul dans `Set` ou `Type`. Et le système de type du Cci garantit cette propriété. Examinons l'exemple suivant :

```
Definition or_carac : ∀A B:Prop, A∨B → nat :=
 fun A B ab ⇒
  match ab with
   | or_introl _ ⇒ 0
   | or_intror _ ⇒ 1
  end.
```

Si cet exemple était légal, pour savoir si une application de `or_carac` se réduit en 0 ou 1, il faudrait calculer le contenu de la preuve logique de `A∨B`, ce qu'on veut précisément éviter. La réponse du système à cette tentative de définition est :

```
Error: Incorrect elimination of ab in the inductive type or.
The elimination predicate fun _:A∨B ⇒ nat has type A∨B → Set.
It should be one of : Prop.

Elimination of an inductive object of sort : Prop
is not allowed on a predicate in sort : Set
because non-informative objects may not construct informative ones.
```

Il existe néanmoins deux exceptions permettant l'élimination d'un inductif sur `Prop` afin de fabriquer un terme informatif. Dans ces deux cas, la forme de l'inductif fait que son filtrage n'apporte aucune information calculatoire.

- Le premier cas est celui d'un inductif logique vide, c'est-à-dire sans constructeur, comme `False`. Cette élimination de `False` va correspondre au principe de « ex-falso quodlibet » : si l'on a un terme de type `False`, c'est que l'on est sous un contexte contradictoire, et on autorise alors la construction d'un terme de n'importe quel type.

```
Definition False_rect: ∀A:Type, False → A :=
 fun (A:Type) (f:False) ⇒ match f with end.
```

- La seconde exception concerne les inductifs singletons logiques. Ce sont les inductifs logiques à un constructeur, et dont le constructeur n'a pas d'arguments informatifs. On peut donner comme exemple `and`, et surtout `eq`. Disposer d'un terme dans un tel type

[2]Lorsque `Set` est imprédicatif, il existe cependant une restriction sur l'élimination de certains inductifs dans `Set` : ils doivent être *petits* pour être autorisés à bâtir un terme de sorte `Type` (cf. section 4.7 de [78])

inductif n'apporte aucun contenu calculatoire, étant donné que l'on sait forcément quel est le constructeur de ce terme, et que les arguments de ce constructeur sont de nouveau sans contenu calculatoire. Voici par exemple le principe de récurrence informatif associé à `eq`, qui en fait décrit comment passer de `(P y)` à `(P x)` si l'on sait que `x=y` :

```
Definition eq_rect: ∀A:Type,∀x:A,∀P:A→Type,P x → ∀y:A,x=y → P y
 := fun (A:Type)(x:A)(P:A→Type)(f:P x)(y:A)(e:x=y) ⇒
     match e with refl_equal ⇒ f end.
```

Ce principe et son pendant logique `eq_ind` sont à la base de la tactique `rewrite` de remplacement de termes par des termes égaux. Il est à noter que ces principes de récurrence `..._rect` et `..._ind` sont engendrés automatiquement par Coq lors de la définition de l'inductif correspondant.

1.1.4 Des termes mixtes `Prop`/`Set`

Jusqu'à maintenant, nous avons vu comment utiliser le Cci comme langage de programmation, et comment calculer avec des termes écrits avec ce Cci. Quel besoin y a-t-il alors d'un mécanisme d'extraction automatique de programmes à partir de termes du Cci, sachant qu'un terme *est* déjà un programme ? Il est en effet vrai que l'extraction de termes purement calculatoires est uniquement un problème de traduction[3] d'un langage d'origine (Cci) vers des langages cibles (Ocaml ou Haskell). Les choses se corsent quand on considère des termes mixtes, avec des parties logiques et calculatoires entrelacées. L'usage de ce style de termes, autorisé par le Cci, se fait sentir dans de nombreuses situations. On peut ainsi enrichir un terme calculatoire avec des pré- et post-conditions, ou encore utiliser une récursion bien fondée en justifiant la décroissance d'une mesure à chaque appel récursif.

Les pré-conditions

Outre le besoin naturel d'exprimer une spécification de fonctions sous la forme pré- et post-conditions, les pré-conditions logiques vont également apporter une solution au problème de la définition de fonctions partielles. Considérons, par exemple, une fonction de division entière `div:nat→nat→nat`, qui n'est pas définie quand son deuxième argument vaut zéro. Il y a alors (au moins) trois façons de procéder :

(i) On peut définir `(div n 0)` malgré tout, à l'aide d'une valeur arbitraire, par exemple `0`. Le problème est alors qu'une division entière peut retourner `0` soit comme résultat légitime, soit comme marque d'une situation anormale. On ne peut donc plus prouver un lemme du genre :

```
Lemma div_gives_zero : ∀n m:nat,(div n m)=0 → n<m.
```

[3]Nous verrons en fait que cette traduction n'est pas si simple que cela, étant donné que le système de types du Cci est beaucoup plus puissant que celui des langages fonctionnels cibles.

(ii) La deuxième solution est de simuler un mécanisme d'exceptions. Cela peut se faire en étendant le domaine d'arrivée `nat` en un $\texttt{nat}_\perp$, comme en théorie des domaines. Plutôt que de modifier ainsi tous les types utilisés, il existe une méthode générique, à savoir l'utilisation d'un inductif `option` :

```
Inductive option (A:Set) : Set :=
 | Some : A → option A
 | None : option A.
```

Ainsi, `div` va retourner `None` si son deuxième argument est `O`, et `(Some r)` sinon, `r` étant alors le vrai résultat du calcul. L'inconvénient de cette méthode est la lourdeur de cet encodage : à chaque utilisation d'une division, il faudra effectuer un filtrage pour soit accéder au véritable résultat, soit de nouveau retourner `None`.

(iii) La dernière possibilité est d'exprimer le fait que le deuxième argument doit être non nul par une pré-condition logique. Le type de `div` devient alors `nat→∀n:nat,n≠O→nat`. On notera que le type du deuxième argument n'est plus donné via un type flèche, c'est-à-dire un produit anonyme, mais par un produit nommé (par `n`), afin de pouvoir y référer dans l'assertion logique. C'est la méthode qui se rapproche le plus de la notion de définition partielle. En effet la fonction `div` n'est pas définie en dehors du domaine de validité de l'assertion logique. La contrepartie est que l'on doit maintenant fournir une preuve logique de non-nullité comme troisième argument à chaque appel à `div`.

```
Coq < Lemma two_non_zero : 2≠0. auto. Qed.
two_non_zero is defined

Coq < Eval Compute in (div 3 2 two_non_zero).
     = 1
     : nat
```

Les post-conditions

Outre les pré-conditions, il est également possible d'exprimer des post-conditions logiques, ce qui en combinant les deux permet de donner la spécification d'une fonction dans un style à la Hoare [45]. Ainsi une fonction de type $A \to B$, de pré-condition P et de post-condition Q correspond à une preuve constructive de la formule :

$$\forall \texttt{x:A},\ \texttt{(P x)} \to \exists \texttt{y:B},\ \texttt{(Q x y)}$$

Nous n'allons pas transcrire cette formule en Coq en utilisant le quantificateur existentiel `ex` sur `Prop` déjà rencontré. En effet, cela reviendrait à considérer le résultat de la fonction comme étant non-calculatoire. Nous allons plutôt utiliser une quantification existentielle calculatoire, nommée `sig`.

```
Inductive sig (A:Set)(P:A→Prop) : Set :=
```

.../...

.../...

```
exist : ∀x:A, P x → sig A P.
```

A noter que `sig A (fun x ⇒ P x)`, qui exprime donc l'existence d'un objet calculatoire `x` vérifiant la propriété logique `(P x)`, s'écrit également `{ x:A | (P x) }`. Ceci nous donne donc la forme générale suivante pour une fonction avec pré- et post-conditions :

$$\forall x{:}A, (P\ x) \rightarrow \{\ y{:}B \mid Q\ x\ y\ \}$$

Par exemple une version avec post-condition de notre fonction de division entière peut se spécifier ainsi :

```
Definition div : ∀a b:nat, b≠0 → { q:nat | q*b ≤ a ∧ a < (S q)*b }.
```

On peut alors retrouver le résultat calculatoire de `div` via un filtrage[4] :

```
Coq < Eval compute in let (q,_) := div 3 2 two_non_zero in q.
     = 1
     : nat
```

Cet inductif `sig` peut également servir à résoudre le problème des fonctions partielles d'une quatrième façon, via une restriction du domaine de départ. On peut ainsi exprimer le type des entiers non-nuls par `{ n:nat | n≠0 }`. Cette approche est équivalente à celle par pré-conditions, et comme elle est légèrement moins naturelle en Coq, nous ne l'utiliserons pas.

La disjonction calculatoire

À côté de `sig`, un autre type fréquemment utilisé combine parties informatives et parties logiques. Il s'agit de `sumbool`, qui est une réplique de la disjonction logique `or`, sauf qu'elle est placé dans l'univers `Set` :

```
Inductive sumbool (A B: Prop) : Set :=
  | left : A → sumbool A B
  | right : B → sumbool A B.
```

Le type `(sumbool A B)` est également noté `{A}+{B}`. La conséquence de l'usage de `Set` dans la définition de `sumbool` est que l'on peut tester si un objet de type `sumbool` commence par `left` ou `right`, y compris lorsque l'on est dans une partie informative. En comparaison, cela serait illégal avec un objet de type `or`. Par contre l'argument de `left` et de `right` reste lui logique, ce qui empêche de l'analyser à des fins constructives. Du point de vue calculatoire, `sumbool` est donc un type très similaire à `bool`, contenant simplement des annotations logiques en plus. Ce type sert donc essentiellement à exprimer des résultats de décidabilité, et peut se lire « il existe un algorithme pour déterminer si `A` ou bien `B` ». Par exemple :

[4] La syntaxe `let ...:=... in ...` de cet exemple est un raccourci pour un filtrage sur un inductif à un seul constructeur. De même une syntaxe `if ... then ... else ...` existe comme raccourci à tout filtrage sur un inductif à deux constructeurs.

```
Theorem eq_nat_dec : ∀n m, {n = m} + {n ≠ m}.
```

Ou encore :

```
Lemma le_lt_dec : ∀n m, {n ≤ m} + {m < n}.
```

La récursion bien-fondée

Les définitions par récursion bien-fondée constituent un dernier exemple d'utilisation de parties logiques dans une fonction calculatoire. Il faut tout d'abord formaliser le fait pour une relation d'être bien-fondée.

```
Section Well_Founded.
 Variable A : Set.
 Variable R : A→A→Prop.
 Inductive Acc : A→Prop :=
  Acc_intro : ∀x:A, (∀y:A, R y x → Acc y) → Acc x.

  Definition Well_founded := ∀a:A, Acc a.
```

L'usage d'une `Section` et de `Variable` permet de factoriser les dépendances communes, ici par rapport à un type `A` et à une relation logique `R` sur ce type. Un élément est dit accessible `(Acc x)` par rapport à `A` et `R` ssi tous les antécédents de `x` par `R` sont eux-mêmes accessibles. Cette définition inductive peut sembler étrange, car sans cas de base. En fait, la quantification universelle fait qu'un élément initial (sans antécédent par `R`) est directement accessible. La finitude des objets inductifs fait qu'on peut interpréter `Acc` ainsi : on a `(Acc x)` si toutes les suites d'antécédents successifs par `R` partant de `x` sont finies. Enfin la relation `R` est dite bien-fondée si tous les points de `A` sont accessibles par `R`.

Nous allons avoir besoin d'une inversion de l'inductif `Acc` : si un point `x` est accessible, alors tous ses antécédents sont accessibles. Ceci s'obtient par filtrage sur `(Acc x)` :

```
Definition Acc_inv : ∀x:A, Acc x → ∀y:A, R y x → Acc y :=
 fun x a ⇒
  match a in Acc x return ∀y, R y x → Acc y with
   | Acc_intro x' f ⇒ f end.
```

Les deux annotations « `in Acc x` » et « `return ∀y, R y x → Acc y` » peuvent être ignorées en première lecture. Elles permettent de préciser quel doit être le type du sous-terme `match ... with ...`. Ces annotations sont facultatives dans les nombreuses situations simples où le système peut inférer un type convenable. C'est ainsi le cas des filtrages rencontrés jusqu'à maintenant, qui produisent des objets de types évident comme `nat`. Mais ici, le type du second argument `f` de `Acc_intro` dépend du premier argument `x'`, qui n'est pas visible de l'extérieur du terme. Il se trouve en fait que `x'` vaut forcément `x`, mais le système ne sait actuellement pas le découvrir seul, d'où le besoin d'une annotation manuelle. Pour une discussion plus générale sur le besoin d'annotations, voir page 84 de [68].

On va maintenant montrer que si un prédicat calculatoire P se propage par R, alors P est valide en tout point accessible.

```
 Section Acc_iter.
  Variable P : A→Type.
  Variable F : ∀x, (∀y, R y x → P y) → P x.
  Fixpoint Acc_iter (x:A)(a:Acc x) {struct a} : P x :=
   F x (fun y h ⇒ Acc_iter y (Acc_inv x a y h)).
 End Acc_iter.
End Well_founded.
```

Il y a deux choses étonnantes dans cette définition Acc_iter :

- Tout d'abord on construit un terme informatif (car de sorte Type) par récurrence sur a, qui est un objet inductif logique. Ceci est légal en Coq, mais peut sembler être une violation du principe sous-jacent à la dualité Prop/Set, selon lequel une partie logique ne doit pas influencer un calcul informatif. En fait cette influence se limite à juste fournir l'assurance que cette récursion va terminer. Le calcul véritable est en fait effectué dans la fonctionnelle F, qui ne peut pas utiliser le contenu de a à cause des restrictions sur les sortes des filtrages.
- Le deuxième point choquant est que l'appel récursif dans (Acc_iter x a) se fait sur l'argument récursif (Acc_inv x a y h), qui ne semble pas structurellement plus petit que l'argument récursif initial a. Mais si a est de la forme (Acc_intro _ f), alors (Acc_inv x a y h) se réduit en (f y h). Comme f est la fonction contenue dans a, cette définition vérifie donc en fait le critère de décroissance structurelle de Coq.

Mentionnons une dernière fonction liée à la bonne fondation :

```
Coq < Check well_founded_induction.
well_founded_induction
     : ∀(A:Set) (R:A→A→Prop),
       well_founded R →
       ∀P:A→Set,
       (∀x:A, (∀y:A, R y x → P y) → P x) →
       ∀a:A, P a
```

Il s'agit d'une variante de Acc_iter, pour laquelle la relation R est supposée bien-fondée, et qui permet donc d'obtenir (P x) pour tout x sans restriction.

Comme application, cette récursion bien-fondée va permettre de définir des fonctions. Considérons par exemple A = nat et R = lt, c'est-à-dire l'ordre strict sur nat. La bibliothèque standard de Coq contient une preuve nommée lt_wf du fait que cet ordre est bien-fondé. On peut alors définir notre division entière div par soustractions successives plutôt que par récurrence structurelle :

```
Definition div : ∀a b:nat, b≠0 → { q:nat | q*b ≤ a ∧ a < (S q)*b }.
```

.../...

```
                                    .../...
Proof.
 intro a; pattern a; apply (well_founded_induction lt_wf); clear a.
 intros a Hrec b Hb.
 elim (le_lt_dec b a); intros Hab.

  assert (H : a-b < a). omega.
  elim (Hrec (a-b) H b Hb); simpl; intros q (Hq,Hq').
  exists (S q); simpl; omega.

  exists 0; omega.
Qed.
```

1.1.5 Extensions de Coq

Un certain nombre d'extensions de Coq n'ont volontairement pas été présentées ici. Elle feront l'objet d'une étude spécifique dans le chapitre 4. Il s'agit en particulier :

- Du nouveau système de modules de Coq (section 4.1)
- Des types co-inductifs (section 4.2)

1.2 Une présentation formelle du Cci

Nous allons maintenant donner une présentation plus formelle du Cci. Ce formalisme sera utilisé dans le chapitre théorique qui va suivre. Les notations utilisées ici correspondent autant que possible à celles du chapitre 4 du Manuel de Référence [78]. La principale exception concerne les environnements et les contextes. Pour des raisons de simplicité, nous allons fusionner ces deux notions. Précisons tout d'abord la syntaxe des termes du Cci.

1.2.1 Syntaxe

Définition 1 (termes) *Les termes du* Cci *sont donnés par la grammaire suivante :*

$$
\begin{array}{rcl}
t & ::= & s \\
 & | & x \mid c \mid C \mid I \\
 & | & \forall x:t,\, t \mid \lambda x:t,\, t \mid \texttt{let}\ x := t\ \texttt{in}\ t \mid (t\ t) \\
 & | & \texttt{case}(t,\, t,\, t\ \ldots\ t) \\
 & | & \texttt{fix}\ x_i\ \{x_1/k_1 : t := t\ \ldots\ x_n/k_n : t := t\}
\end{array}
$$

Avec :

- *s désigne une sorte, parmi* Set, Prop *ou* $\texttt{Type}_i$. *Nous omettrons l'indice de* $\texttt{Type}_i$ *tant qu'il n'intervient pas explicitement.*
- *x, c, C, I sont des identificateurs, qui réfèrent chacun à des variables, des constantes, des constructeurs inductifs et des types inductifs.*
- *pour le point-fixe, les k_i désignent des entiers, qui vont correspondre au nombre d'arguments attendus par les composantes x_i.*

Pour alléger les syntaxes, nous utiliserons parfois des notations vectorielles :

- $(f\ \overrightarrow{x})$ pour $(f\ x_1\ \ldots\ x_n)$
- $\forall \overrightarrow{x : X}, T$ pour $\forall x_1 : X_1, \ldots \forall x_n : X_n, T$
- $\lambda \overrightarrow{x : X}, t$ pour $\lambda x_1 : X_n, \ldots \lambda x_n : X_n, t$

Et nous utiliserons la notation $|\overrightarrow{x}|$ pour préciser la taille du vecteur $\overrightarrow{x}$, lorsque cette taille sera importante et non évidente.

On remarquera que la syntaxe choisie ici diffère quelque peu de la syntaxe concrète Coq :

- Le λ est plus concis que le mot-clé `fun`.
- La syntaxe `match ... with`, même si elle est un progrès au niveau du confort d'utilisation, n'est guère adaptée à un raisonnement théorique, surtout en présence des annotations supplémentaires `as`, `in` et `return`. Nous utiliserons à la place une syntaxe `case`$(P, e, \overrightarrow{f})$ dans laquelle e est l'objet filtré, P est le prédicat d'élimination et les fonctions $\overrightarrow{f}$ correspondent aux branches mises sous forme fonctionnelle[5] : l'équation $(C\ x\ y\ z) \Rightarrow t$ donne la fonction $\lambda x, \lambda y, \lambda z, t$. Signalons au passage qu'un filtrage peut parfaitement n'avoir aucune branche, ce que nous noterons `case`$(e, P, \varnothing)$. Quant au prédicat P, si e est un objet inductif de type $(I\ \overrightarrow{q}\ \overrightarrow{u})$, alors les règles de typage qui vont suivre imposent à P d'être de la forme $\lambda \overrightarrow{u}, \lambda x : (I\ \overrightarrow{q}\ \overrightarrow{u}), P_0$. L'équivalent en syntaxe concrète Coq est alors :

  `match e as x in` $(I\ \overrightarrow{\_}\ \overrightarrow{u})$ `return` P_0 `with ... end`
- La syntaxe concrète Coq permet de définir un bloc de fonctions récursives mutuelles anonymes via :

  `fix` f_1`:=`t_i `with ... with` f_n`:=`t_n `in` f_i

  Nous utiliserons ici la syntaxe abrégée suivante :

  `fix` $f_i\ \{f_1/k_1 : T_1 := t_1\ \ldots\ f_n/k_n : T_n := t_n\}$

  avec pour chaque composante f_i, l'entier k_i indiquant le rang de l'argument inductif sur lequel s'effectue la récursion. Cet entier correspond à l'annotation `struct` de la syntaxe concrète Coq. Nous allons nommer cet argument inductif la « garde », car il va être utilisé pour contrôler la réduction du point-fixe (voir les réductions plus bas). On parle alors de récursion gardée.

Définition 2 (contextes) *Un contexte Γ est une liste pouvant contenir comme éléments :*

- *des hypothèses $(x : t)$*
- *des définitions $(c := t : t')$*
- *des déclarations d'inductifs $\mathtt{Ind}_n(\Gamma_I := \Gamma_C)$, avec n étant le nombre de paramètres, et les contextes Γ_I et Γ_C contenant respectivement les types inductifs et leurs constructeurs.*

[5] Pour la petite histoire, cette syntaxe est analogue à celle en vigueur dans les versions 5.x. Elle était encore utilisable dans les versions 7.x via le mot-clé `Case ... of`.

Par rapport aux notations du Manuel de Référence, on a choisi de ne pas détacher les paramètres dans un contexte à part, mais de les laisser apparaître à la fois dans Γ_I et Γ_C. L'annotation n permet de signaler que les n premiers produits dans tous les éléments de Γ_I et Γ_C correspondent à des paramètres. C'est d'ailleurs ce qui est fait dans l'implantation. Par exemple, écrivons les déclarations des entiers unaires et des listes polymorphes dans cette syntaxe :

$\texttt{Ind}_0(\texttt{nat} : \texttt{Set} := \texttt{O} : \texttt{nat};\ \texttt{S} : \texttt{nat} \to \texttt{nat})$
$\texttt{Ind}_1(\texttt{list}:\texttt{Set}\to\texttt{Set} :=$
$\quad \texttt{nil} : \forall A{:}\texttt{Set}, \texttt{list}\ A;\ \texttt{cons} : \forall A{:}\texttt{Set}, A \to \texttt{list}\ A \to \texttt{list}\ A)$

1.2.2 Réductions

Définition 3 (réductions) *Les réductions du* Cci *sont les suivantes :*

(*beta*) $((\lambda x : X,\ t)\ u) \to_\beta t\{x \leftarrow u\}$

(*delta*) $c \to_\delta t$ *si le contexte courant* Γ *contient* $(c := t : T)$.

(*zeta*) $\texttt{let}\ x := t\ \texttt{in}\ u \to_\zeta u\{x \leftarrow t\}$

(*iota*) $\texttt{case}(C_i\ \overrightarrow{p}\ \overrightarrow{u},\ P,\ f_1\ \ldots\ f_n) \to_\iota f_i\ \overrightarrow{u}$
lorsque C_i *est le* i*-ème constructeur d'un type inductif ayant* $|\overrightarrow{p}|$ *paramètres.*

(*iota*) *Soit* F *le bloc récursif* $f_1/k_1 : A_1 := t_1\ \ldots\ f_n/k_n : A_n := t_n$. *Alors :*
$(\texttt{fix}\ f_i\ \{F\}\ u_1\ \ldots\ u_{k_i}) \to_\iota (t_i\{f_j \leftarrow \texttt{fix}\ f_j\ \{F\}\}_{\forall j}\ u_1\ \ldots\ u_{k_i})$
si u_{k_i}*, l'argument « de garde », commence par un constructeur.*

Ces réductions sont *fortes* : elles sont autorisées à n'importe quelle position interne d'un terme, via les règles habituelles de compatibilité. Par opposition, nous aurons aussi à considérer par la suite des réductions *faibles*, c'est-à-dire des réductions ne pouvant intervenir qu'au sommet du terme, ou bien à gauche ou à droite d'une application ou bien en tête d'un `Case`, bref en dehors de tout lieu. Nous utiliserons $\to_r$ pour désigner un pas de n'importe laquelle des réductions fortes $\to_\beta$, $\to_\delta$, $\to_\iota$ ou $\to_\zeta$.

À partir de ces réductions, on définit alors la relation de convertibilité $=_{\beta\delta\iota\zeta}$ et l'ordre de cumulativité $\leq_{\beta\delta\iota\zeta}$ qui vont servir dans la règle de typage (Conv).

Définition 4 (convertibilité) *Deux termes* u *et* v *sont convertibles (noté* $u =_{\beta\delta\iota\zeta} v$*) s'ils possèdent un réduit commun* w*, c'est-à-dire tel que* $u \to_r^* w$ *et* $v \to_r^* w$.

Définition 5 (cumulativité) *L'ordre de cumulativité* $\leq_{\beta\delta\iota\zeta}$ *est défini récursivement par :*

- *Si* $u =_{\beta\delta\iota\zeta} v$ *alors* $u \leq_{\beta\delta\iota\zeta} v$
- $\texttt{Type}_i \leq_{\beta\delta\iota\zeta} \texttt{Type}_j$ *dès que* $i \leq j$
- $\texttt{Set} \leq_{\beta\delta\iota\zeta} \texttt{Type}$
- $\texttt{Prop} \leq_{\beta\delta\iota\zeta} \texttt{Type}$
- *Si* $T =_{\beta\delta\iota\zeta} T'$ *et* $U \leq_{\beta\delta\iota\zeta} U'$ *alors* $\forall x : T,\ U \leq_{\beta\delta\iota\zeta} \forall x : T',\ U'$

Notons qu'à cause de la δ-réduction, ces deux notions dépendent implicitement d'un contexte.

Définition 6 (arité) *Une arité est un terme convertible à une sorte ou à un produit* $\forall x : T, U$ *avec* U *de nouveau une arité. Une arité peut donc s'écrire après réduction sous la forme* $\forall x_1 : X_1, \ldots \forall x_n : X_n, s$*. On parlera alors d'une arité de sorte* s*.*

1.2.3 Typage

Nous allons maintenant donner une définition condensée des règles de typages du Cci, pour mémoire. Encore une fois, nous renvoyons au Manuel de Référence [78] pour une version détaillée et commentée de ces règles.

Définition 7 (typage) *Le jugement de typage* $\Gamma \vdash t : T$*, qui signifie que* T *est un type valide pour* t *sous le contexte* Γ*, est défini en même temps que la propriété* $\mathcal{WF}(\Gamma)$ *de bonne formation d'un contexte, le tout via les règles de la figure 1.1.*

Détaillons maintenant les conditions auxiliaires des règles de typage des inductifs :

1. Dans la règle (Prod), la condition $\mathcal{P}(s_1, s_2, s_3)$ sur les sortes est :
   $(s_2 = s_3 = \texttt{Prop}) \vee$
   $(s_2 = s_3 = \texttt{Set} \wedge s_1 \neq \texttt{Type}) \vee$
   $(s_1 = \texttt{Type}_i \wedge s_2 = \texttt{Type}_j \wedge s_3 = \texttt{Type}_k \wedge i \leq k \wedge j \leq k)$
   Remarquons que pour rendre (de nouveau) `Set` imprédicatif, il suffit d'enlever la condition $s_1 \neq \texttt{Type}$ de la deuxième ligne.
2. Dans la règle (I-WF) de bonne formation d'une définition inductive, $\mathcal{I}_n(\Gamma_I, \Gamma_C)$ regroupe toutes les conditions auxiliaires qui doivent être remplies pour que cette définition soit valide :
   - Tous les noms contenus dans Γ_I et Γ_C doivent être distincts et nouveaux.
   - Comme nous n'avons pas explicité les paramètres, mais seulement leur nombre n, il faut s'assurer que toutes les déclarations de Γ_I et Γ_C commencent par les mêmes n produits $\forall \overrightarrow{p : P}$, et que toutes les occurrences des inductifs I dans Γ_C sont appliquées au moins à $\overrightarrow{p}$.
   - Pour tout $(I : A) \in \Gamma_I$, A doit être une arité sur une sorte s_I.
   - Pour tout $(C : T) \in \Gamma_C$, T doit être un type de constructeur pour l'un des types inductifs I défini dans Γ_I, *i.e.* T doit être de la forme $\forall \overrightarrow{p : P}, \forall \overrightarrow{x : X}, I\ \overrightarrow{p}\ \overrightarrow{y}$. De plus la sorte s_C dans la prémisse de typage de T doit être s_I.
   - T doit également vérifier la condition de positivité vis-à-vis de tous les types de Γ_I. Cette condition est essentielle pour garantir la normalisation forte, mais n'intervient pas dans l'extraction. Nous ne la détaillerons donc pas.
3. Dans la règle (Case), la substitution σ remplace les n paramètres formels $\overrightarrow{p}$ par n paramètres concrets $\overrightarrow{q}$. Quant à la condition $\mathcal{C}(I\ \overrightarrow{q} : A_\sigma; B)$, elle exprime le fait que le type d'arrivée du `case` doit être compatible avec l'inductif I qui est filtré :
   - On a $\mathcal{C}(I : (\forall x : X,\ A); (\forall x : X,\ B))$ ssi pour tout x, on a $\mathcal{C}(I\ x : A; B)$.
   - On a $\mathcal{C}(I : \texttt{Prop}; I \rightarrow \texttt{Prop})$

$$\mathcal{WF}(\varnothing) \qquad \frac{\Gamma \vdash T : s \quad x \notin \Gamma}{\mathcal{WF}(\Gamma; (x : T))} \qquad \frac{\Gamma \vdash t : T \quad c \notin \Gamma}{\mathcal{WF}(\Gamma; (c := t : T))} \tag{WF}$$

$$\frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \texttt{Set} : \texttt{Type}_i} \qquad \frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \texttt{Prop} : \texttt{Type}_i} \qquad \frac{\mathcal{WF}(\Gamma) \quad i < j}{\Gamma \vdash \texttt{Type}_i : \texttt{Type}_j} \tag{Ax}$$

$$\frac{\mathcal{WF}(\Gamma) \quad (x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\mathcal{WF}(\Gamma) \quad (c := t : T) \in \Gamma}{\Gamma \vdash c : T} \tag{Var)(Cst}$$

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma; (x : T) \vdash U : s_2 \quad \mathcal{P}(s_1, s_2, s_3)}{\Gamma \vdash \forall x : T,\, U : s_3} \tag{Prod}$$

$$\frac{\Gamma \vdash \forall x : U,\, T : s \quad \Gamma; (x : U) \vdash t : T}{\Gamma \vdash \lambda x : U,\, t : \forall x : U,\, T} \qquad \frac{\Gamma \vdash t : \forall x : U,\, T \quad \Gamma \vdash u : U}{\Gamma \vdash (t\ u) : T\{x \leftarrow u\}} \tag{Lam)(App}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma; (x := t : T) \vdash u : U}{\Gamma \vdash \texttt{let}\ x\ :=\ t\ \texttt{in}\ u : U\{x \leftarrow u\}} \tag{Let}$$

$$\frac{\Gamma \vdash U : s \quad \Gamma \vdash t : T \quad T \leq_{\beta\delta\iota\zeta} U}{\Gamma \vdash t : U} \tag{Conv}$$

$$\frac{\mathcal{WF}(\Gamma) \quad \texttt{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (I : A) \in \Gamma_I}{\Gamma \vdash I : A} \tag{I-Type}$$

$$\frac{\mathcal{WF}(\Gamma) \quad \texttt{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (C : T) \in \Gamma_C}{\Gamma \vdash C : T} \tag{I-Cons}$$

$$\frac{\begin{array}{c}\text{pour tout } (I : A) \in \Gamma_I, \quad \Gamma \vdash A : s \\ \text{pour tout } (C : T) \in \Gamma_C, \quad \Gamma; \Gamma_I \vdash T : s_C \\ \mathcal{I}_n(\Gamma_I, \Gamma_C)\end{array}}{\mathcal{WF}(\Gamma; \texttt{Ind}_n(\Gamma_I := \Gamma_C))} \tag{I-WF}$$

$$\frac{\begin{array}{c}\texttt{Ind}_n(\Gamma_I = \Gamma_C) \in \Gamma \quad (I : \forall \overrightarrow{p : T},\, A) \in \Gamma_I \quad \sigma = \{\overrightarrow{p} \leftarrow \overrightarrow{q}\} \\ |\overrightarrow{p}| = n \quad \Gamma \vdash e : I\ \overrightarrow{q}\ \overrightarrow{u} \quad \Gamma \vdash P : B \quad \mathcal{C}(I\ \overrightarrow{q} : A_\sigma; B) \\ \text{pour tout constructeur } C_i : \forall \overrightarrow{p : T},\, \forall \overrightarrow{x : X},\, I\ \overrightarrow{p}\ \overrightarrow{y}, \\ \Gamma \vdash f_i : \forall \overrightarrow{x : X_\sigma},\, P\ \overrightarrow{y_\sigma}\ (C_i\ \overrightarrow{q}\ \overrightarrow{x})\end{array}}{\Gamma \vdash \texttt{case}(e,\, P,\, f_1 \ldots f_m) : P\ \overrightarrow{u}\ e} \tag{Case}$$

$$\frac{\forall i,\ \Gamma \vdash A_i : s_i \quad \forall i,\ \Gamma; \overrightarrow{(f : A)} \vdash t_i : A_i \quad \mathcal{F}(\overrightarrow{f}, \overrightarrow{A}, \overrightarrow{k}, \overrightarrow{t})}{\Gamma \vdash \texttt{fix}\ f_j\ \{f_1/k_1 : A_1 := t_1\ \ldots\ f_n/k_n : A_n := t_n\} : A_j} \tag{Fix}$$

Fig. 1.1: Règles de typage de Cci

- On a $\mathcal{C}(I : \texttt{Prop}; I \to s)$ pour une sorte $s \neq \texttt{Prop}$ ssi I est un inductif logique vide ou singleton.
- On a $\mathcal{C}(I : \texttt{Set}; I \to s)$ pour toute sorte s.
- On a $\mathcal{C}(I : \texttt{Type}; I \to s)$ pour toute sorte s.

Dans la définition précédente :

- Un inductif logique vide est un inductif de sorte `Prop` avec zéro constructeur.
- Un inductif logique singleton est un inductif de sorte `Prop` avec un seul constructeur dont tous les arguments non-paramètres sont de sorte `Prop`.

Notons que si l'on prend `Set` imprédicatif, il ne faut accepter $\mathcal{C}(I : \texttt{Set}; I \to \texttt{Type})$ que lorsque I est un petit inductif, c'est-à-dire dont aucun constructeur n'a d'argument non-paramètre de sorte `Type`.

4. Dans la règle (Fix), la condition $\mathcal{F}(\overrightarrow{f}, \overrightarrow{A}, \overrightarrow{k}, \overrightarrow{t})$ impose :
   - pour tout i, A_i doit être de la forme $\forall \overrightarrow{x : X}, A'_i$, avec au moins k_i produits, et le type X_{k_i} du k_i-ème produit doit être un type inductif.
   - de plus t_i ne peut contenir que des appels récursifs décroissants : si f_j apparaît dans t_i, alors il doit avoir au moins k_j arguments, et son k_j-ème argument doit être structurellement plus petit que l'argument inductif d'origine x_{k_i}. La définition précise de ce « structurellement plus petit » se trouve dans le Manuel de Référence. Informellement, il s'agit de dire que tout sous-terme d'un terme inductif obtenu en traversant au moins un constructeur est structurellement plus petit que le terme de départ.

1.2.4 Propriétés

Une première propriété de Cci est que si l'on a $\Gamma \vdash t : T$, alors il existe une sorte $s \in \{\texttt{Prop}, \texttt{Set}, \texttt{Type}\}$ telle que $\Gamma \vdash T : s$. Tout terme t bien typé admet donc au moins une sorte s.

Remarquons ensuite qu'on ne peut pas parler en toute rigueur « du » type et de « la » sorte d'un terme t, car il n'y a pas unicité des types dans le Cci. Par exemple un objet de type `Prop` admet également le type `Type` via la règle (Conv) et la cumulativité.

Cependant, un terme ne peut pas admettre à la fois `Prop` et `Set` comme sorte. On peut donc parler de la sorte la plus petite d'un terme, vis-à-vis de l'ordre de cumulativité ($\texttt{Prop} \leq \texttt{Type}$ et $\texttt{Set} \leq \texttt{Type}$). Cette sorte la plus petite sera aussi nommée *sorte principale*.

De même tous les types d'un terme sont comparables dans l'ordre de cumulativité. On peut même montrer qu'il existe un type, unique modulo conversion, qui est plus petit que tous les autres types vis-à-vis de l'ordre de conversion. On le nommera *type principal*. En fait, cette notion n'a d'intérêt que pour des types qui sont des arités. Dans les autres cas, tous les types possibles sont égaux modulo conversion. Et on parlera alors parfois, par abus, « du » type.

Par rapport à un contexte Γ, un terme T est un type s'il admet pour type une sorte s dans ce contexte. En fait, plutôt que ces types Coq, c'est un sur-ensemble de ceux-ci qui va jouer un rôle crucial pour l'extraction :

Définition 8 (schéma de types) *Un schéma de types est un terme bien-typé dont au moins un type est une arité, c'est-à-dire a la forme* $\forall x_1 : X_1, \ldots \forall x_n : X_n, s$ *avec* s *une sorte.*

En d'autres termes, un schéma de types est un terme qui va devenir un type quand il sera appliqué à suffisamment d'arguments. Par exemple $\lambda X : \texttt{Type}, X \to X$ est un schéma de types : une fois appliqué à un type, on obtient le type flèche correspondant. À l'inverse $\lambda X : \texttt{Type}, \lambda x : X, x$ n'est pas un schéma de types : il se peut qu'on obtienne un type en l'appliquant (par exemple à `Set` et `nat`), mais on peut également ne pas obtenir un type (par exemple par l'application à `nat` et `O`).

Lemme 1 (stabilité) *Le* Cci *admet les résultats suivants :*

1. (*Subject Reduction*) *Soit un terme* t *se réduisant vers* u*. Alors tout type* T *de* t *est également un type de* u*. Et toute sorte* s *de* t *est également une sorte de* u*.*
2. *D'autre part, lors d'une substitution d'une variable dans un terme, le type de ce terme peut évidemment changer. Plus précisément, si* T *est un type de* t*, alors* $T\{x \leftarrow u\}$ *est un type de* $t\{x \leftarrow u\}$*. Mais les propriétés suivantes sont conservées malgré tout :*
   - *Si* t *a pour sorte* `Prop`*, il en est de même pour* $t\{x \leftarrow u\}$
   - *Si* t *est un schéma de types, il en est de même pour* $t\{x \leftarrow u\}$
   - *Si* t *a un type inductif, il en est de même pour* $t\{x \leftarrow u\}$
3. *Enfin, en ce qui concerne les applications, si* t *a pour sorte* `Prop`*, il en est de même pour* $(t\ u)$*, et si* t *est un schéma de types, alors* $(t\ u)$ *l'est également.*

Preuve. Nous admettrons ici ces résultats. Se reporter sinon à des études théoriques du Cci, comme par exemple [83]. □

Il est à noter que pour chacune de ces propriétés de stabilité, leurs réciproques sont fausses :

1. Prenons un type quelconque dans `Prop`, disons `True`. Alors $((\lambda X : \texttt{Type}, X)\ \texttt{True})$ peut se réduire en `True`. Or ce dernier terme accepte le type `Prop`, et pas le premier. Et dans cet exemple, $\texttt{Type}_0$ est une sorte de `True`, mais non du terme initial, qui n'a que $\texttt{Type}_1$ comme sorte principale.
2. - Plaçons-nous dans le contexte $\Gamma = (X : \texttt{Type})$. Alors $(\lambda x : X, x)\{X \leftarrow \texttt{True}\}$ admet `Prop` comme sorte alors que $\lambda x : X, x$ a `Type` pour sorte principale.
   - Dans ce même contexte, $(\lambda x : X, x)\{X \leftarrow \texttt{Set}\}$ est un schéma de types alors que $\lambda x : X, x$ n'en était pas un à l'origine.
   - Prenons maintenant $\Gamma = (b : \texttt{bool})$. Soit $T = \texttt{case}(b, \texttt{Type}, \texttt{nat}, \texttt{bool})$. Alors le type $t = \texttt{case}(b, T, \texttt{O}, \texttt{true})$ n'admet pas de type inductif, alors que $t\{b \leftarrow \texttt{true}\}$ a pour type $T\{b \leftarrow \texttt{true}\}$ qui est convertible à `nat`.
3. Soit $\texttt{Id} = \lambda X : \texttt{Type}, \lambda x : X, x$. Ce n'est ni un terme de sorte `Prop`, ni un schéma de types. Et pourtant $(\texttt{Id}\ \texttt{True})$ a pour sorte `Prop`, et $(\texttt{Id}\ \texttt{Set})$ est un schéma de types.

Dans tous les cas, les contre-exemples font intervenir la sorte `Type`, ce qui illustre donc les précautions à prendre pour concevoir une extraction capable de gérer `Type`. Ceci étant dit, il faut néanmoins relativiser l'importance des problèmes de changement de type principal lors de la réduction :

Lemme 2 *Soit t un terme bien typé de* Cci *se réduisant vers u, et soient T et U les types principaux respectifs de t et u.*

(i) *On a $U \leq_{\beta\delta\iota\zeta} T$.*

(ii) *Si t n'est pas un schéma de types, $T =_{\beta\delta\iota\zeta} U$.*

(iii) *t est un schéma de types ssi u est un schéma de types.*

(iv) *t et u ont même sorte principale (si l'on regarde les différentes sortes* `Type`$_i$ *comme une seule).*

Preuve.

(i) Il suffit de faire remarquer que par « subject reduction », T est toujours un type de u. Dès lors, la propriété de principalité de U nous donne le résultat souhaité.

(ii) Si t n'est pas un schéma de types, cela signifie que T n'est pas une arité. Or on peut avoir $U \leq_{\beta\delta\iota\zeta} T$ sans que $U =_{\beta\delta\iota\zeta} T$ que si ces deux types sont des arités avec des sortes finales $s_U < s_T$.

(iii) $U \leq_{\beta\delta\iota\zeta} T$ implique que :

- soit U et T sont égaux modulo $\beta\delta\iota\zeta$, et en particulier sont conjointement des arités ou non.
- soit U et T sont distincts modulo $\beta\delta\iota\zeta$, tous les deux des arités, avec des sortes finales $s_U < s_T$.

(iv) Passons à la preuve dans le premier cas : si t et u sont des schémas de types, alors ils ont pour sorte principale `Type`. Sinon ils partagent exactement les mêmes types (modulo $\beta\delta\iota\zeta$), et donc les mêmes sortes.

□

La conservation des sortes principales lors de la réduction (iv) semble être invalidée par le premier des contre-exemples précédents. Ce principe est-il alors vraiment correct ? Venant d'un Normand, il est normal que la réponse soit oui et non. Oui, si l'on s'intéresse juste à la distinction `Set`/`Prop`/`Type`, et que l'on oublie les index des `Type`$_i$. Et non dans le cas contraire, comme le montre bien le contre-exemple. En fait le point de vue de l'extraction va être le premier.

On retrouve ainsi une frontière entre les termes logiques et informatifs : un terme logique (*i.e.* de sorte principale `Prop`) ne peut devenir informatif (*i.e.* de sorte principale `Set` ou `Type`) au cours d'une réduction, et réciproquement.

Nous aurons également à étudier les formes possibles des termes Cci clos et en forme normale.

Lemme 3 *Soit t un terme bien typé de* Cci*, clos et en forme normale modulo $\beta\delta\iota\zeta$.*

1. *Si t possède un type inductif $(I\ \overrightarrow{v})$, il commence par un constructeur de ce type I.*
2. *Si t possède un type produit, il est :*
   a. *soit un constructeur inductif partiellement appliqué*
   b. *soit un type inductif qui peut être seul ou bien appliqué à des arguments, y compris de façon partielle.*

   c. *soit une λ-abstraction*
   d. *soit un point-fixe* ($\mathtt{fix}\ f_i\ \{\ldots\}\ \overrightarrow{u}$). *Et dans ce cas, si l'argument de « garde » est le* k_i*-ème, on a alors* $|\overrightarrow{u}| < k_i$.

3. *Si* t *possède une sorte* s *comme type (*i.e. t *est un type), il est :*
   a. *soit une sorte*
   b. *soit un type inductif complètement appliqué*
   c. *soit un produit*

Preuve. On raisonne par récurrence sur le typage de t, et par cas sur la dernière règle :

- Tout d'abord, la règle (Var) ne peut produire de terme clos, et les règles (Cst) et (Let) fabriquent des termes non-normaux.
- La règle (Ax) correspond au cas (3a).
- la règle (Prod) correspond au cas (3b).
- (Lam) : On ne peut pas être dans les parties (1) ou (3) de l'énoncé, et la partie (2c) est clairement vérifiée.
- (App) : Prenons $t = (t'\ u)$. L'hypothèse de récurrence sur $\Gamma \vdash t' : \forall x : U, T$ nous donne quatre cas :
  a. $t' = (C\ \overrightarrow{v})$. Alors $t = (C\ \overrightarrow{v}\ u)$. Si C est encore partiellement appliqué dans t, alors t possède un type produit, et on est dans le cas (2a) de l'énoncé. Si maintenant C est complètement appliqué, t possède un type inductif : le cas (1) de l'énoncé est bien vérifié.
  b. $t' = (I\ \overrightarrow{v})$. Alors t est juste un type inductif un peu plus appliqué. Si ce type inductif est maintenant complètement appliqué, il a pour type une sorte, et on est dans le cas (3b). Sinon son type est toujours un produit, et on est dans le cas (2b).
  c. t' ne peut être une λ-abstraction, sinon $(t'\ u)$ serait réductible.
  d. t' est alors un point-fixe manquant d'arguments. Supposons que u soit l'argument de « garde » attendu par le point-fixe. Ce u est alors un terme inductif, et l'hypothèse de récurrence concernant $\Gamma \vdash u : U$ montre qu'il commence par un constructeur. Ce point-fixe est donc réductible, ce qui est contradictoire avec les suppositions initiales. On est donc bien toujours en présence d'un point-fixe manquant d'arguments. Enfin, ce manque d'arguments implique nécessairement que ce point fixe a un type produit. Les parties (1) et (3) de l'énoncé sont donc exclues, et la partie (2) est correcte via le cas (2d).
- (Conv) : On utilise directement l'hypothèse de récurrence.
- (I-Type) : Si $\Gamma \vdash I : A$ et que l'arité A possède au moins un produit, la situation (2b) est vérifiée. Et si A est directement une sorte, on est dans le cas (3b).
- (I-Cons) : On est soit dans la situation (1) si le constructeur n'attend aucun argument, soit dans la situation (2a) sinon.

- (Case) : L'argument de récurrence concernant la tête du `case`, qui est un terme inductif, montre que ce terme commence par un constructeur. Le `case` est donc réductible, ce qui est absurde.
- (Fix) : Un point-fixe sans argument en attend toujours au moins un (l'argument de garde). On ne peut donc être dans les parties (1) et (3) de l'énoncé. Par contre, la partie (2d) de l'énoncé est clairement validée.

□

Curieusement, l'énoncé même de ce lemme montre qu'il n'y a pas d'autres cas à considérer comme type T de t. En effet, T peut également être choisi clos et en forme normale. La partie (3) de l'énoncé affirme alors qu'il s'agit d'une sorte, d'un produit ou d'un type inductif.

Il est aussi à noter que la preuve de ce résultat n'exige l'absence de redex qu'à des endroits précis, à savoir au sommet, à gauche et à droite d'une application et en tête d'un `case`. Ce résultat va donc rester parfaitement valide quand nous étudierons la réduction faible du Cci.

1.2.5 Cci$_m$: une variante de Cci adaptée à l'étude sémantique

La présentation du Cci donnée jusqu'ici va convenir à la partie la plus syntaxique de l'étude théorique de l'extraction. Par contre, dans un second temps, nous allons réaliser une étude plus sémantique de la correction de l'extraction dans la section 2.4. Et dans cette optique, le Cci ainsi formulé va être légèrement inadapté. Nous présentons ici la variante Cci$_m$ que nous utiliserons alors.

Le principal souci dans cette partie 2.4 va être que Cci autorise la promotion silencieuse d'une proposition au rang de type informatif. Ainsi `True`, originellement de type `Prop`, peut être également vu avec le type `Type`, via la règle de typage (Conv). Ceci va être gênant, puisque l'on va désirer attribuer une sémantique différente à une proposition et à un type informatif, et ce sans forcément avoir à considérer le typage pour être renseigné.

Une solution est alors d'utiliser une marque au niveau de la syntaxe pour signaler l'emploi d'une cumulativité `Prop` $\leq$ `Type`. Ainsi `True`† va désigner l'objet `True` promu au niveau `Type`. Cette technique de marquage s'inspire de nos travaux de DEA [54], eux-mêmes reprenant l'idée de [28]. Pour des raisons similaires, quoique moins fondamentales, il sera également intéressant de marquer la promotion d'un objet `Set` au niveau `Type`. Ainsi `nat`‡ va désigner le type `nat` vu au niveau `Type`. Au niveau de la syntaxe Coq, ces marques se traduiraient par des « casts » explicites. Ainsi `True`† et `nat`‡ s'écriraient `(Prop_Type True)` et `(Set_Type nat)`, avec :

```
Definition Prop_Type (t:Prop) : Type := t.
Definition Set_Type (t:Set) : Type := t.
```

Avant de poursuivre la présentation de ce système Cci$_m$ avec marques, signalons tout d'abord que nous allons quelque peu restreindre la portée de la cumulativité dans ce système, pour des raisons de simplicité de la présentation. Par exemple, la règle (Conv) d'origine

autorise à écrire :

$$\frac{\Gamma \vdash t : \texttt{nat} \to \texttt{Set} \quad (\texttt{nat} \to \texttt{Set}) \leq_{\beta\iota} (\texttt{nat} \to \texttt{Type})}{\Gamma \vdash t : \texttt{nat} \to \texttt{Type}}(\text{Conv})$$

Désormais nous n'allons autoriser l'emploi de la cumulativité que sur les types et non sur les schémas de types : on accepte $\texttt{Set} \leq \texttt{Type}$ mais plus $\texttt{nat} \to \texttt{Set} \leq \texttt{nat} \to \texttt{Type}$. Ceci interdit alors l'exemple précédent, mais on peut s'en approcher via une η-expansion :

$$\frac{\dfrac{\dfrac{\Gamma \vdash t : \texttt{nat} \to \texttt{Set}}{\Gamma; (x : \texttt{nat}) \vdash (t\ x) : \texttt{Set}}(\text{App})}{\Gamma; (x : \texttt{nat}) \vdash (t\ x) : \texttt{Type}}(\text{Conv})}{\Gamma \vdash \lambda x : \texttt{nat}, (t\ x) : \texttt{nat} \to \texttt{Type}}(\text{Lam})$$

Pour une étude plus détaillée sur la cumulativité dans les théories des types d'ordres supérieures, on pourra consulter [57].

Revenons maintenant à nos marques † et ‡. Il s'agit d'annotations pouvant porter sur tout terme de Cci. Et nous remplaçons la règle (Conv) par quatre nouvelles règles (Conv) (CumT) (CumS) (CumP), afin de rendre obligatoire la présence de marques après usage de la cumulativité $\texttt{Prop} \leq \texttt{Type}$ ou $\texttt{Set} \leq \texttt{Type}$:

$$\frac{\Gamma \vdash U : s \quad \Gamma \vdash t : T \quad T =_{\beta\iota\delta\zeta} U}{\Gamma \vdash t : U}(\text{Conv}) \qquad \frac{\Gamma \vdash t : \texttt{Type}_i \quad i < j}{\Gamma \vdash t : \texttt{Type}_j}(\text{CumT})$$

$$\frac{\Gamma \vdash t : \texttt{Set}}{\Gamma \vdash t^\ddagger : \texttt{Type}}(\text{CumS}) \qquad \frac{\Gamma \vdash t : \texttt{Prop}}{\Gamma \vdash t^\dagger : \texttt{Type}}(\text{CumP})$$

La cumulativité de $\texttt{Type}_i$ vers $\texttt{Type}_j$, sans conséquence pour l'extraction, reste elle implicite. On remarquera qu'un terme marqué bien typé ne peut avoir deux marques successives : $t^\dagger$ étant de type Type, il est impossible de former $t^{\dagger\dagger}$.

Il est important de signaler que les marques sont ignorées par les règles de typages lorsqu'elles apparaissent à droite du jugement. Seul nous importe en effet l'influence de la marque sur le typage d'un terme. Par exemple l'application suivante est bien légale :

$$\frac{\Gamma \vdash t : \forall x : U^\dagger, T \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T\{x \leftarrow u\}}(\text{App})$$

Il serait possible de rendre formel cet aspect du typage, en ajoutant les quatre règles suivantes :

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t : T^\dagger} \qquad \frac{\Gamma \vdash t : T^\dagger}{\Gamma \vdash t : T} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash t : T^\ddagger} \qquad \frac{\Gamma \vdash t : T^\ddagger}{\Gamma \vdash t : T}$$

En pratique, ces règles resteront implicites tout au long de notre étude.

Précisons maintenant le statut de ces marques (par exemple †) vis-à-vis de la substitution :

- Une substitution sans effet ne change rien aux marques, par exemple $y\{x \leftarrow t^\dagger\} = y$
- Bien sûr, une substitution sans marque n'en crée pas : $x\{x \leftarrow t\} = t$
- Une variable remplacée par un terme marqué donne un terme marqué : $x\{x \leftarrow t^\dagger\} = t^\dagger$

- Une variable marquée donne un terme marqué après substitution : $x^\dagger\{x \leftarrow t\} = t^\dagger$
- Enfin, le cas $x^\dagger\{x \leftarrow t^\dagger\}$ est interdit par le typage. En effet, $x^\dagger$ est nécessairement de type `Type`, et x de type `Prop`. Or le $t^\dagger$ que l'on veut lui substituer est aussi dans `Type`, ce qui rend la substitution impossible.

Quant aux réductions, elles vont influer sur les marques uniquement via des substitutions. A priori, on pourrait avoir besoin de préciser ce que devient une marque à cheval sur un redex, comme dans $(\lambda x : U,\, t)^\ddagger\ u$. Mais en fait, ce cas est exclu car un terme marqué est forcément un type, qui ne peut débuter par une λ-abstraction. De même, la tête d'un ι-redex ne peut être marquée car elle possède un type inductif. Ce n'est donc pas un type.

Nous devons maintenant justifier de nouveau un certain nombre de propriétés que notre marquage pourrait avoir rendu caduques. Commençons par la propriété de « subject reduction », et avant cela par la préservation du typage par substitution.

Lemme 4 *Soit t et u deux termes de Cci_m admettant pour types respectifs T et U. Si x est une variable libre de t de type U, alors $t\{x \leftarrow u\}$ admet $T\{x \leftarrow u\}$ pour type dans le Cci_m.*

Preuve. Par récurrence sur la dérivation de typage de t. □

Lemme 5 *Soit t un terme de Cci_m admettant T comme type, et u un réduit de t. Alors u admet également T comme type.*

Preuve. Il s'agit de la même preuve que pour Cci, mais avec le lemme précédent comme résultat de substitution. □

Maintenant, le principal effet de la présence des marques est l'unicité des types :

Lemme 6 *Soit t un terme de Cci_m admettant T et U comme types. Alors $T =_{\beta\delta\iota\zeta} U$, à condition de ne pas distinguer ici les différentes sortes $\texttt{Type}_i$ entre elles, et d'ignorer les marques sur les types T et U.*

Preuve. Dans Cci, la non-unicité des types venait de la possibilité d'utiliser la partie cumulativité de la règle (Conv) à tout moment. Mais avec la présence des marques, les règles (CumS) et (CumP) sont maintenant des règles dont l'usage est contrôlé par la syntaxe du terme, comme la plupart des autres règles. Quant à la règle (CumT), elle ne modifie par les types, du moins selon le point de vue que nous avons choisi.

Une fois ceci remarqué, on procède par récurrence sur la dérivation du typage $t : T$, et en comparant cette dérivation avec celle de $t : U$. Hormis les $\beta\delta\iota\zeta$-conversions (Conv) que nous ignorerons, ces deux dérivations ont nécessairement la même forme. □

En particulier la notion de type principal devient désormais triviale. Reprenons par exemple le contre-exemple qui montrait que le type principal dans Cci n'était pas forcément conservé lors d'une réduction : $((\lambda X : \texttt{Type},\, X)\ \texttt{True})$. Il est maintenant mal typé dans Cci_m. Pour en faire un terme valide, il faudrait ajouter une marque : $((\lambda X : \texttt{Type},\, X)\ \texttt{True}^\dagger)$. Mais son réduit $\texttt{True}^\dagger$ n'admet alors plus comme type que `Type`.

Pour finir cette présentation de Cci_m, nous allons expliciter les traductions possibles entre Cci et Cci_m. Dans le sens le plus simple, il est immédiat de prendre un terme t de

type T dans CCI_m et d'obtenir un terme correspondant bien typé dans CCI : il suffit d'enlever les marques. Et T (moins ses marques) reste alors un type valide dans CCI. Dans le sens réciproque, en partant d'un terme t dans CCI ayant pour type principal T, on peut obtenir un terme correspondant bien typé dans CCI_m et qui ne diffère de t que par la présence de marques (et éventuellement d'η-expansion, cf. la restriction sur la cumulativité). Introduire ces marques est facile : il suffit de prendre une dérivation de typage de $t : T$, et d'adapter les utilisations de la règle (Conv) en une seule passe de haut en bas. De la sorte, le type du terme CCI_m obtenu est encore T (ou plutôt sa version avec marques).

Chapitre 2

L'extraction des termes Coq

Ce chapitre est consacré à la présentation de notre fonction d'extraction $\mathcal{E}$ sur les termes Cci. Pour l'instant, cette fonction va retourner en sortie des termes d'un langage théorique intermédiaire nommé $\text{Cci}_{\Box}$, non typé, qui va jouer ici un rôle similaire à celui de F_{ω} dans l'ancienne extraction. Le prochain chapitre s'occupe ensuite de la traduction vers le langage final, et en particulier du problème du typage du code extrait dans ce langage.

Outre la présentation de cette fonction $\mathcal{E}$, ce chapitre contient également la preuve de correction de $\mathcal{E}$, ou plutôt *les* preuves de correction, étant donné que cette étude théorique est divisée en deux parties, à savoir :

- Une première étude plutôt syntaxique nous permet de garantir le bon déroulement de toute réduction de termes extraits, que l'on utilise une évaluation stricte ou bien paresseuse. Cette partie est une version révisée des résultats présentés dans [55], eux-mêmes inspirés de [54].
- Dans un deuxième temps, nous établirons dans la section 2.4 la correction de ces termes extraits vis-à-vis des spécifications originelles Coq au moyen d'un argument sémantique inspiré de la réalisabilité.

Mais dans un premier temps, nous allons détailler les limitations de l'ancienne fonction $\mathcal{E}$ de C. Paulin, qui nous ont conduit à la version actuelle de $\mathcal{E}$.

2.1 Les difficultés de l'élimination des parties logiques

Un danger potentiel de la suppression des parties logiques est que cette suppression peut modifier l'ordre d'évaluation. Prenons par exemple une fonction de type $\forall$`x:A,(P x)`$\rightarrow$ `B`, avec `A` et `B` des types informatifs, et `P` une propriété logique. Notre fonction `f` attend donc un argument informatif `x` et une preuve que cet argument satisfait la pré-condition `P x`. Si l'on dispose également d'un terme `t` de type `A` et d'une preuve `p` de type `(P t)`, on peut alors former les deux termes Coq bien typés `(f t)` et `(f t p)`. Au niveau Coq, ces deux termes sont de nature sensiblement différente. Par exemple, l'évaluation de `(f t)` sera sans doute vite bloquée par le manque du second argument `p`, alors que `(f t p)` est une application totale qui peut normalement se réduire vers une valeur de type `B` (si par exemple ces termes `f`, `t` et `p` sont clos).

Examinons maintenant l'action d'une fonction $\mathcal{E}$ d'extraction qui supprime complètement les parties logiques, comme le fait l'ancienne extraction. Les deux termes précédents `(f t)` et `(f t p)` s'extraient alors vers le même terme `(`$\mathcal{E}$`(f) `$\mathcal{E}$`(t))`, car `p`, logique, disparaît. Et comme le type de $\mathcal{E}$`(f)` est alors de la forme $\mathcal{E}$`(A)`$\rightarrow$$\mathcal{E}$`(B)`, le terme extrait `(`$\mathcal{E}$`(f) `$\mathcal{E}$`(t))`, qui est une application totale, se comporte plutôt comme `(f t p)`, et donc peut se réduire complètement. Ceci est par contre un comportement bien différent de celui de `(f t)`.

Cette modification de l'ordre d'évaluation a, bien sûr, une influence sur l'efficacité de l'exécution de termes extraits. Mais elle peut même être fatale au bon déroulement de cette exécution. L'ancienne extraction pouvait en particulier engendrer des termes extraits dont l'évaluation s'arrêtait prématurément sur une exception non prévue, ou tout autre erreur d'exécution. Et à l'inverse ou pouvait également rencontrer des termes dont l'évaluation ne terminait pas.

L'exemple typique de code pouvant lever une exception en cas d'extraction naïve fait intervenir la constante `False_rec`. Ce terme Coq de type $\forall$`P:Set, False`$\rightarrow$`P` est utilisé pour traiter les sous-cas absurdes d'une preuve. Par exemple, la tactique `contradiction` de Coq génère des termes de preuves avec cette constante. Ainsi, lorsque l'on définit une fonction `f` de type $\forall$`x:nat,(x`$\neq$`0)`$\rightarrow$`nat`, on peut donc se servir de cette constante dans le cas où `x=0`. Lors de l'extraction, ce `False_rec` est traduit en une exception, qui signifie que l'exécution ne doit jamais arriver dans ce sous-cas absurde. Maintenant, reprenons notre application partielle `(f 0)`, légale en Coq, qui ne recevra normalement jamais de second argument de type `0`$\neq$`0`. L'extraction `(`$\mathcal{E}$`(f) 0)` va alors s'exécuter sans attendre l'argument logique disparu, et va alors lever l'exception associée à `False_rec`. Notre nouvelle extraction résout ce problème en restituant à l'extraction de `(`$\mathcal{E}$`(f) 0)` son statut de clôture. Pour cela on va laisser des abstractions artificielles `fun _ → ...` chaque fois que nécessaire.

Nous verrons par la suite dans la partie 2.3.2 qu'il est possible également d'engendrer d'autres types d'erreurs d'exécution en combinant ancienne extraction, application partielle et certaines constantes Coq comme `eq_rec`. Quant aux exemples de code extraits ne terminant pas avec l'ancienne extraction, ils sont basés sur la constante `Acc_rec`, qui offre la possibilité en Coq de définir un point-fixe informatif justifié par une mesure de décroissance logique. Il est alors possible, sous un contexte absurde, de fournir une fausse justification logique. Le point-fixe extrait, débarrassé de cette justification logique, peut alors boucler.

Une catégorie de limitations de l'ancienne extraction concerne les univers Coq. En effet, la distinction en Coq entre les parties informatives et logiques est rendu floue par la présence de l'univers `Type`. On peut, en effet, former des termes hybrides comme `if b then nat else True`, où `b` est un booléen. Ce terme va être soit informatif soit logique selon la valeur du booléen `b`. Cette construction n'est possible que par l'existence de règles dites de cumulativité, qui expriment que `Type` contient au moins `Set` et `Prop`. Dans notre exemple, `nat:Set` implique donc également `nat:Type` et de même `True:Prop` implique `True:Type`. Et finalement, notre terme hybride est bien typable avec le type `Type`. L'extraction précédente refusait tout simplement d'extraire un tel type, et plus généralement tout terme utilisant plus ou moins directement la sorte `Type`. Cette restriction drastique permettait alors une élimination complète des parties logiques (du moins dans un système interdisant `False_rec`, `eq_rec` et `Acc_rec`). A l'opposé, le but de notre travail étant de pouvoir traiter tout terme

Coq, nous allons alors devoir utiliser une constante ad hoc (que l'on va noter $\Box$) pour marquer les emplacements précédemment occupés par des parties logiques, comme le `True` ci-dessus. Notre approche est donc similaire aux méthodes par élagages [12, 15].

2.2 La nouvelle fonction d'extraction $\mathcal{E}$

Cette fonction $\mathcal{E}$ d'extraction va éliminer tout sous-terme de sorte `Prop`, car ces sous-termes correspondent à des parties logiques, comme expliqué au chapitre précédent. Mais en plus de cela, nous allons aussi éliminer les sous-termes correspondant à des types, et plus généralement à des schémas de types. Pourquoi supprimer ces schémas de types ? Ce choix est moins naturel que celui conduisant à éliminer les parties de sorte `Prop`. En particulier, il existe au moins un développement Coq dont le résultat principal est la construction du type d'un treillis particulier [62]. L'extraction d'un tel développement ne va alors donner qu'une constante arbitraire remplaçant ce type. Mais cette situation est exceptionnelle. Dans les développements usuels, les résultats concernent des types de données, comme par exemple des types inductifs tels que `bool`, `nat` ou `Z`. Et dans ces cas, nous allons montrer que les schémas de types correspondent à du code mort du point de vue du calcul. Une autre justification de ce choix d'élimination est qu'à la différence de Coq, nos langages cibles (Ocaml et Haskell) font une distinction claire entre le niveau des types et celui des termes, et en particulier ne permettent pas l'usage des types comme des termes ordinaires.

Nous définissons Cci$_\Box$ à partir du même langage que Cci, avec en plus une constante spéciale $\Box$. Par contre les termes de Cci$_\Box$ seront non typés. En effet, nous n'adapterons pas la relation de typage du Cci à Cci$_\Box$. Enfin, les réductions dans Cci$_\Box$ sont définies comme étant exactement celles de Cci, avec $\Box$ vu comme une constante non-réductible.

Définissons maintenant notre fonction d'extraction de Cci vers Cci$_\Box$.

Définition 9 (fonction $\mathcal{E}$) *La fonction d'extraction $\mathcal{E}$ est définie par récurrence structurelle sur tout terme t typable dans un contexte Γ :*

($\Box$) *Si t est un schéma de types ou admet* `Prop` *comme sorte dans le contexte Γ, alors $\mathcal{E}_\Gamma(t) = \Box$*

Sinon, on discrimine selon la structure de t :

(id) $\mathcal{E}_\Gamma(a) = a$ *si a est une variable c, une constante c ou un constructeur C.*

(lam) $\mathcal{E}_\Gamma(\lambda x : T,\, t) = \lambda x : \Box,\, \mathcal{E}_{\Gamma'}(t)$ *où* $\Gamma' = \Gamma; (x : T)$

(let) $\mathcal{E}_\Gamma(\texttt{let}\ x := t\ \texttt{in}\ u) = \texttt{let}\ x := \mathcal{E}_\Gamma(t)\ \texttt{in}\ \mathcal{E}_{\Gamma'}(u)$ *où* $\Gamma' = \Gamma; (x := t : T)$ *et T est un type de t*

(app) $\mathcal{E}_\Gamma(u\ v) = (\mathcal{E}_\Gamma(u)\ \mathcal{E}_\Gamma(v))$

($cases$) $\mathcal{E}_\Gamma(\texttt{case}(e,\, P,\, f_1\ \ldots\ f_n)) = \texttt{case}(\mathcal{E}_\Gamma(e),\, \Box,\, \mathcal{E}_\Gamma(f_1)\ \ldots\ \mathcal{E}_\Gamma(f_n))$

(fix) $\mathcal{E}_\Gamma(\texttt{fix}\ f_i\ \{f_1/k_1 : A_1 := t_1\ \ldots\ f_n/k_n : A_n := t_n\}) =$
$\texttt{fix}\ f_i\ \{f_1/k_1 : \Box := \mathcal{E}_{\Gamma'}(t_1)\ \ldots\ f_n/k_n : \Box := \mathcal{E}_{\Gamma'}(t_n)\}$
où $\Gamma' = \Gamma; (f_1 : A_1); \ldots; (f_n : A_n)$

Et l'extraction d'un contexte est définie par :

$$(nil)\ \ \mathcal{E}(\varnothing) = \varnothing$$
$$(def)\ \ \mathcal{E}(\Gamma; (c := t : T)) = \mathcal{E}(\Gamma); (c := \mathcal{E}_\Gamma(t) : \Box)$$
$$(ax)\ \ \mathcal{E}(\Gamma; (x : T)) = \mathcal{E}(\Gamma); (x : \Box)$$
$$(ind)\ \ \mathcal{E}(\Gamma; \mathtt{Ind}_n(\Gamma_I := \Gamma_C)) = \mathcal{E}(\Gamma); \mathtt{Ind}_n(\mathcal{E}(\Gamma_I) := \mathcal{E}(\Gamma_C))$$

Clairement, $\mathcal{E}$ est une fonction « d'élagage » : elle ne fait que remplacer certains sous-termes par $\Box$. En particulier il n'y a aucune modification de la structure. Dans l'extraction actuellement implantée en Coq, une seconde phase est dédiée à des modifications de la structure. Cette phase sera décrite dans le chapitre 4. Cet « élagage » est différent de ce que faisaient les extractions précédentes de Coq, qui en particulier ne retiraient pas les types, et supprimaient complètement les λ-abstractions logiques, via une règle du genre :

(lam') $\mathcal{E}(\lambda x : P,\, t) = \mathcal{E}(t)$ *si* P *admet* `Prop` *pour type.*

En terme de réalisabilité, cette dernière règle (lam') correspond à la réalisabilité modifiée, alors que notre nouvelle règle (lam) correspond plus à la réalisabilité récursive. Mais comme nous l'avons expliqué dans la section précédente, la règle (lam') n'est pas correcte si on la combine avec l'évaluation stricte à la Ocaml.

À noter également l'absence de règle dédiée explicitement au produit, puisqu'un produit est toujours un type, et donc a fortiori un schéma de types. La règle $(\Box)$ s'applique donc.

2.3 Étude syntaxique de la réduction des termes extraits

Dans un premier temps, nous allons étudier la réduction des termes extraits, et en particulier prouver que cette réduction est nécessairement finie. Ceci va se faire au moyen d'une méthode syntaxique : nous allons simuler les dérivations des termes extraits par celles des termes Coq d'origine.

Mais cette première approche, relativement simple, est peu appropriée pour établir des propriétés de correction plus sémantique, en particulier lorsque des valeurs fonctionnelles et/ou des termes non clos sont en jeu. La section 2.4 sera alors consacrée à une étude complémentaire, basée sur une extension de la notion de réalisabilité.

En ce qui concerne l'étude syntaxique qui suit, elle se décompose elle-même en deux parties. La section 2.3.1 aboutit au théorème 1 qui établit la forte normalisation des termes extraits, mais uniquement dans une version légèrement bridée du Cci. Ensuite la section 2.3.2 traite du Cci dans son intégralité, ce qui oblige par contre à se restreindre à l'étude des réductions faibles de termes extraits. Dans ce cas, le résultat principal de correction est alors le théorème 5.

2.3.1 Réduction forte dans une restriction de Cci$_\Box$

Nous désirons ici établir que l'évaluation d'un terme extrait termine, et que le résultat de cette évaluation a un sens, par exemple `true` ou `false` pour un terme original dans Cci de type `bool`. Et bien sûr, nous souhaitons également que ce résultat soit cohérent avec la réponse que donnerait l'évaluation du terme original dans Cci.

Nous allons donc procéder par simulation dans Cci des dérivations possibles dans $\text{Cci}_\Box$ et vice-versa. Le problème est alors que cette simulation peut aboutir à un terme du Cci comportant encore des redex, alors que son analogue dans $\text{Cci}_\Box$ n'est plus réductible. En fait, il y a trois catégories potentielles de redex du Cci correspondant à des zones non-redex de $\text{Cci}_\Box$:

1. un β-redex $(\lambda x : X,\, t)\; u$ correspondant à un non-redex $\Box\; u'$
2. un ι-redex $\texttt{case}(e, \ldots, \ldots)$ correspondant à un non-redex $\texttt{case}(\Box, \ldots, \ldots)$
3. un ι-redex $(\texttt{fix}\; f_i\; \{\ldots\}\; u_1 \ldots\; u_n)$ correspondant à un non-redex, qui peut être :
   a. soit $(\Box\; u'_1\; \ldots\; u'_n)$
   b. soit $(\texttt{fix}\; f_i\; \{\ldots\}\; u_1\; \ldots\; \Box)$ (la « garde » est maintenant un $\Box$ bloquant la réduction)

Dans le cas 1, nous aimerions avoir $(\lambda x : X,\, t)\; u$ correspondant à $\Box$ directement au lieu de $\Box\; u$. En effet le lemme de stabilité affirme que l'application conserve le fait d'être de sorte `Prop` ou un schéma de types. Si $\lambda x : X,\, t$ a pu devenir $\Box$, alors il devrait donc « moralement » en être de même pour $(\lambda x : X,\, t)\; u$. Ce « manque de précision » d'un terme extrait peut en fait apparaître après quelques étapes de réductions, comme le montre l'exemple suivant :

Exemple 1

$$t = (\lambda X : \texttt{Type},\, \lambda f : \texttt{nat}{\to}X,\, \lambda g : X \to \texttt{nat},\, (g\; (f\; \texttt{0})))\;\; \texttt{Prop}\;\; (\lambda\_,\, \texttt{True})$$
$$\mathcal{E}(t) = (\lambda X : \Box,\, \lambda f : \Box,\, \lambda g : \Box,\, (g\; (f\; \texttt{0})))\;\; \Box\;\; \Box$$
$$\to^*_\beta\;\; \lambda g : \Box,\, (g\; (\Box\; \texttt{0}))$$

Si besoin est, nous allons résoudre ce problème du cas 1 (en même temps de celui du cas 3a) grâce à une réduction ad hoc :

Définition 10 ($\Box$-réduction) *La $\Box$-réduction est définie par la règle* $(\Box\; u) \to_\Box \Box$

La situation du cas 2 est assez différente. À la différence de l'exemple précédent où un lambda devenait logique après réduction, un filtrage ne peut pas changer le type de l'inductif sur lequel il est effectué. Et $\mathcal{E}$ élimine tous les filtrages produisant des objets de sortes `Prop`. Comme normalement un filtrage d'un inductif sur `Prop` ne peut bâtir qu'un objet de nouveau dans `Prop`, le cas 2 ne devrait normalement pas se produire. Mais cette restriction sur les filtrages logiques possède deux exceptions, concernant les inductifs logiques vides et les inductifs singletons logiques (voir le chapitre précédent). Par exemple Cci autorise les dérivations suivantes :

$$\frac{p : \texttt{False} : \texttt{Prop} \quad T : \texttt{Set}}{\texttt{case}(p,\, T,\, \varnothing)\;\; : T} \qquad \frac{p : x = y : \texttt{Prop} \quad q : P\; x : \texttt{Set}}{\texttt{case}(p,\, P,\, q)\;\; : P\; y : \texttt{Set}}$$

La première dérivation correspond à la constante `Coq` nommée `False_rec`, tandis que la seconde correspond à `eq_rec`.

Plus généralement, une élimination `case` logique peut produire quelque chose d'informatif si l'élimination est effectuée sur un terme dont le type inductif comporte :

1. soit zéro constructeur (inductif vide, comme `False` en `Coq`)
2. soit un seul constructeur dont tous les arguments sont logiques, mis à part éventuellement les paramètres (inductif singleton logique, comme `eq`)

Ceci est en fait une première exception au slogan : « les objets logiques n'entrent jamais en ligne de compte lors de calculs d'objets informatifs ». La seconde entorse à ce principe est le cas 3b : la « garde » d'un point-fixe peut être un terme inductif logique alors que le point-fixe complet est informatif.

Supposons un moment que ces particularités du typage Coq soient désactivées. Jusqu'à la fin de cette section 2.3.1, nous allons considérer deux systèmes Cci$^-$ et Cci$^-_\square$ qui sont Cci et Cci$_\square$ avec les restrictions suivantes :

(i) L'élimination d'inductifs logiques vides ne peut pas produire de termes informatifs.

(ii) Il en est de même pour l'élimination d'inductifs singletons logiques.

(iii) Pour toute composante f_i d'un point-fixe, sa « garde » ne doit pas être logique à moins que le type de f_i ne le soit également.

Nous allons maintenant comparer les résultats respectifs des réductions dans Cci$^-_\square$ et Cci$^-$. Pour simplifier cette comparaison, nous n'allons parler ici que des types sans contenu logique :

Définition 11 *Un type T de* Cci *est dit sans contenu logique si pour toute forme normale close t de type T nous avons $\mathcal{E}(t) = t$.*

Les types de données usuels, comme `bool` ou `nat` vérifient cette condition. Bien sûr, un objet non-réduit dans un tel type peut contenir des parties logiques, mais elles vont disparaître lors de la réduction pour finir sur `true` ou `(S (S O))` par exemple. Nous avons alors le résultat suivant concernant la réduction forte des termes extraits dans ces types sans contenu logique :

Théorème 1 *Soit t un terme clos et bien-typé de* Cci$^-$ *dont un type T est non-logique. Alors toute réduction de $\mathcal{E}(t)$ termine sur la forme normale (dans* Cci$^-$*) de t.*

Nous ne prouverons pas ce résultat, car il est d'importance moindre que le théorème 5 de la section suivante, tandis que les preuves des deux théorèmes sont similaires. De même, il est possible d'établir une version plus générale de ce résultat, traitant également des types qui ne sont pas sans contenu logique. Mais comme la forme normale de $\mathcal{E}(t)$ peut alors contenir des $\square$, il faut de nouveaux outils pour la comparer avec la forme normale de t. Là encore, ceci sera développé dans la section suivante.

Enfin, notons que l'usage de $\square$-réductions n'est pas nécessaire dans ce résultat précis. Ceci s'explique par la conjonction des restrictions (i), (ii) et (iii) avec l'hypothèse supposant que T est non-logique. Par contre les résultats de la section suivante devront utiliser cette $\square$-réduction.

2.3.2 Réduction faible dans le Cci$_\square$ complet

Comme notre objectif est un mécanisme d'extraction acceptant tous les termes Coq, nous devons désormais supprimer ces restrictions (i), (ii) et (iii). La restriction (i) concernant un inductif vide est en fait facile à supprimer, étant donné qu'une ι-réduction sur un inductif vide ne va en fait jamais se produire, faute de constructeur pour déclencher la réduction. Nous pouvons donc juste ignorer ces `case`, et les traduire plus tard vers des exceptions (voir

l'étude de `False_rec` en section 2.1). Par contre la suppression des restrictions (ii) et (iii) va nous obliger à adapter les réductions autorisées sur les termes extraits, en abandonnant la réduction forte (*i.e.* possible sous les lambdas) au profit de la réduction faible. De toute façon, nos langages fonctionnels cibles n'autorisent pas la réduction forte. Centrer notre étude sur la réduction faible est donc parfaitement légitime.

L'élimination singleton

Si H est une égalité (donc logique), `case`(H, `nat`, `O`) peut être réduit et donner `O` même sans connaître la valeur exacte de H, cachée derrière un $\Box$. De façon analogue, nous pouvons réduire systématiquement toute élimination d'un inductif singleton logique. Mais cela est dangereux en combinaison avec la réduction forte, et peut mener à des erreurs d'exécution. Considérons par exemple la fonction `cast` suivante qui transforme un entier en un booléen à condition de pouvoir prouver que les entiers et les booléens coïncident[1] :

```
Definition cast : (nat=bool) → nat → bool :=
 fun (H:nat=bool)(n:nat) ⇒
  match H in (_=bool) return bool with
   | refl_equal ⇒ n
  end.
```

Prenons alors l'exemple suivant :

```
Definition exemple :=
 fun (H:nat=bool) ⇒
  let b : bool := cast H 0 in
  match b with
   | true ⇒ 0
   | false ⇒ 1
  end.
```

Si maintenant on effectue la réduction a priori du `case` dans cet exemple, le sous-terme `(cast H 0)` se réduirait vers l'entier 0, alors que le `match` qui suit attend un booléen, ce qui va entraîner une erreur d'exécution.

Un exemple similaire peut également amener l'entier `0` à être considéré comme étant une fonction si l'on dispose en hypothèse de l'égalité `nat = (nat→nat)`. Et si l'on applique cette « fonction » `0`, on peut se retrouver en cas de réduction forte des termes extraits avec une erreur d'exécution comme `(0 0)`.

Clairement, si l'on interdit la réduction sous les lambdas, ces problèmes disparaissent. En effet, un terme inductif singleton hors des lambdas est forcément clos, et peut donc toujours se réduire vers un constructeur, ce qui légitime notre réduction des éliminations de singletons logiques hors des lambdas.

[1] Nous utiliserons ici la syntaxe `Coq`, plus lisible. L'équivalent dans notre syntaxe théorique de ce `match` avec annotations est `case`(H, (λt : `Set`, λH : (`eq Set nat` t), t), n)

Les points-fixes avec des « gardes » logiques

Le problème est maintenant de réduire un point-fixe informatif dont l'argument servant de « garde » est logique. Bien sûr, la tentation immédiate est de supprimer cette condition de « garde », au moins pour cette catégorie de points-fixes. Mais ceci combiné à la réduction forte peut aboutir à une évaluation qui ne termine pas. La fonction `loop` suivante est bâtie sur le modèle de `Acc_iter` (voir le chapitre précédent). Elle attend une preuve hypothétique de l'énoncé faux affirmant l'accessibilité de 0 par la relation `gt` (c'est-à-dire $>$ sur les entiers naturels de Coq). Si cette preuve était fournie, `loop` se lancerait alors dans une infinité d'appels récursifs : `F n` appellerait `F (S n)` et ainsi de suite.

```
Definition loop :=
 fun (Ax:Acc gt 0) ⇒
  (fix F (n:nat)(a:Acc gt n) {struct a} : nat :=
    F (S n) (Acc_inv a (S n) (gt_Sn_n n)))
  0 Ax.
```

Le sous-terme `(Acc_inv a ...)` est une preuve de l'accessibilité de `(S n)`, utilisant les constantes `Acc_inv` et `gt_Sn_n` fournies par la bibliothèque standard de Coq. L'extraction $\mathcal{E}$ donne alors :

$$\begin{aligned}\mathcal{E}(\texttt{loop}) = \;& \lambda Ax : \square, \\ & \texttt{fix}\ F\ \{F/2 : \square := \\ & \quad \lambda n : \texttt{nat}, \lambda a : \square, (F\ (\texttt{S}\ n)\ \square)\} \\ & \texttt{O}\ \square\end{aligned}$$

Et si l'on retire ici la condition « de garde », alors ce terme peut se réduire fortement même sans être appliqué, et donner $\lambda Ax : \square$, `fix` $F\ \{\ldots\}$ (`S O`) $\square$, et ainsi de suite ...

Modification de la réduction

Pour gérer ces cas exceptionnels de ι-réduction sur des termes logiques, nous devons tout d'abord ajouter une annotation supplémentaire sur les termes du Cci. Les `match` portant sur des inductifs à un seul constructeur s'écrivent normalement en Coq :

`match` e `with C` $\overrightarrow{x}$ $\Rightarrow$ `t end`

La syntaxe « fonctionnelle » utilisée dans cette étude théorique, à savoir :

`case`$(e, \ldots, (\lambda\overrightarrow{x}, t))$

présente l'inconvénient de perdre le nombre d'arguments $\overrightarrow{x}$ du constructeur unique `C` de notre inductif. En effet, avec cette syntaxe, il n'est pas correct de compter le nombre de lambdas, car `t` peut en contenir d'autres. Nous allons remédier à ce problème en marquant en indice ce nombre d'arguments pour ces `case` à branche unique :

$\texttt{case}_n(e, \ldots, (\lambda\overrightarrow{x}, t))$

Et bien sûr, la fonction d'extraction $\mathcal{E}$ va garder ces annotations. Pour ne pas (plus) alourdir les notations, nous omettrons parfois ces annotations dans les situations où elles ne sont pas utilisées.

Voici maintenant les modifications à apporter aux réductions de $\text{CCI}_\Box$ afin de pouvoir gérer ces $\Box$ bloquant des ι-réductions.

Définition 12 (nouvelle ι-réduction) *La ι-réduction sur des termes de $\text{CCI}_\Box$ est désormais :*

(*iota*) $\texttt{case}(C_i\ \overrightarrow{p}\ \overrightarrow{u},\ P,\ f_1\ \ldots\ f_m) \to_\iota f_i\ \overrightarrow{u}$

(*iota*) $\texttt{case}_n(\Box,\ P,\ f) \to_\iota f\ \underbrace{\Box\ \ldots\ \Box}_{n}$

(*iota*) *Soit F le bloc récursif $f_1/k_1 : A_1 := t_1\ \ldots\ f_n/k_n : A_n := t_n$. Alors :*
$(\texttt{fix}\ f_i\ \{F\}\ u_1\ \ldots\ u_{k_i}) \to_\iota (t_i\{f_j/\texttt{fix}\ f_j\ \{F\}\}_{\forall j}\ u_1\ \ldots\ u_{k_i})$
lorsque u_{k_i} est $\Box$ ou commence par un constructeur.

Nous allons également restreindre les réductions en interdisant la réduction forte : pour chaque réduction possible, on lui associe une réduction faible.

Définition 13 (réductions faibles) *Les réductions $\to_{\beta_w}$, $\to_{\iota_w}$, $\to_{\delta_w}$, $\to_{\zeta_w}$ et $\to_{\Box_w}$ sont définies à partir des mêmes règles de base que respectivement $\to_\beta$, $\to_\iota$, $\to_\delta$, $\to_\zeta$, $\to_\Box$, mais en ajoutant seulement une partie des règles de compatibilité, à savoir :*

$$\frac{u \to_? v}{(u\ t) \to_? (v\ t)} \qquad \frac{u \to_? v}{(t\ u) \to_? (t\ v)}$$

$$\frac{u \to_? v}{\texttt{case}(u,\ P,\ \ldots) \to_? \texttt{case}(v,\ P,\ \ldots)}$$

Enfin, comme pour $\to_r$, la réduction faible complète $\to_{r_w}$ est $\to_{\beta_w} \cup \to_{\iota_w} \cup \to_{\delta_w} \cup \to_{\zeta_w}$.

En fait, cette réduction $\to_{r_w}$ peut être vue comme une généralisation commune des stratégies d'appels par valeur et par nom d'Ocaml et de Haskell. La dernière étape vers la réduction réellement implantée dans ces langages est de fixer un ordre d'évaluation. Réduire d'abord les arguments va nous donner la stratégie stricte de Ocaml. Et à l'inverse réduire tout d'abord en tête du terme correspond à la stratégie paresseuse de Haskell.

Un point important à mentionner est que toute cette étude théorique va se faire dans des contextes sans axiome. En effet, étudier la réduction en présence d'axiomes revient à étudier la réduction forte sous les lambdas correspondants à ces axiomes, ce que nous souhaitons justement éviter. En particulier nous utiliserons à plusieurs reprises la propriété fondamentale selon laquelle un terme inductif clos dans un contexte sans axiome se réduit forcément vers un terme commençant par un constructeur. La présence d'un axiome peut suffire à invalider cette propriété. Bien sûr, tous les axiomes n'ont pas cet effet, mais pour des raisons de simplicité, nous les interdisons tous.

Pour étudier l'évaluation des termes extraits, nous allons la comparer à l'évaluation des termes d'origine. Il nous faut alors un invariant liant termes d'origine et termes extraits et stable par réduction. Il est alors tentant d'utiliser directement la fonction $\mathcal{E}$ pour cet invariant. Malheureusement, cela n'est pas un bon choix, car $\mathcal{E}$ se comporte mal vis-à-vis de la réduction : si $t \to_r u$, on peut ne pas avoir $\mathcal{E}(t) \to_r \mathcal{E}(u)$ dans certains cas[2]. À la place

[2]Le terme t de l'exemple 1 est un tel contre-exemple. En effet, $\mathcal{E}(t)$ peut se réduire vers un terme contenant $(\Box\ \texttt{0})$, alors que $\mathcal{E}$ ne produira jamais un tel sous-terme, mais directement $\Box$.

de $\mathcal{E}$, on va donc définir et utiliser une relation non-déterministe $\rightarrow_{\mathcal{E}}$ qui va avoir de bonnes propriétés d'invariance.

Définition 14 (relation $\rightarrow_{\mathcal{E}}$) *La relation non-déterministe $\rightarrow_{\mathcal{E}}$ reliant un terme de* Cci *et un terme de* Cci$_\square$*, et dépendant implicitement d'un contexte Γ, est définie par les règles suivantes :*

$$\frac{\Gamma \vdash t : \forall \overrightarrow{x : X},\ s}{t \rightarrow_{\mathcal{E}} \square}\ (\mathcal{E}\text{–}\square_1) \qquad \frac{\Gamma \vdash t : T : \texttt{Prop}}{t \rightarrow_{\mathcal{E}} \square}\ (\mathcal{E}\text{–}\square_2)$$

$$\frac{a = x \text{ ou } a = c \text{ ou } a = C}{a \rightarrow_{\mathcal{E}} a}\ (\mathcal{E}\text{–}id) \qquad \frac{t \rightarrow_{\mathcal{E}} t'}{\lambda x : T,\ t \rightarrow_{\mathcal{E}} \lambda x : \square,\ t'}\ (\mathcal{E}\text{–}lam)$$

$$\frac{t \rightarrow_{\mathcal{E}} t' \quad u \rightarrow_{\mathcal{E}} u'}{\texttt{let } x := t \texttt{ in } u \rightarrow_{\mathcal{E}} \texttt{let } x := t' \texttt{ in } u'}\ (\mathcal{E}\text{–}let) \qquad \frac{t \rightarrow_{\mathcal{E}} t' \quad u \rightarrow_{\mathcal{E}} u'}{(t\ u) \rightarrow_{\mathcal{E}} (t'\ u')}\ (\mathcal{E}\text{–}app)$$

$$\frac{e \rightarrow_{\mathcal{E}} e' \quad \forall i,\ f_i \rightarrow_{\mathcal{E}} f'_i \quad \textit{Info}(e)}{\texttt{case}(e,\ P,\ f_1\ \ldots\ f_n) \rightarrow_{\mathcal{E}} \texttt{case}(e',\ \square,\ f'_1\ \ldots\ f'_n)}\ (\mathcal{E}\text{–}case)$$

$$\frac{\forall i,\ t_i \rightarrow_{\mathcal{E}} t'_i}{\texttt{fix}\, f_i\, \{f_1/k_1 : A_1 := t_1 \ldots f_n/k_n : A_n := t_n\} \rightarrow_{\mathcal{E}} \texttt{fix}\, f_i\, \{f_1/k_1 : \square := t'_1 \ldots f_n/k_n : \square := t'_n\}}\ (\mathcal{E}\text{–}fix)$$

La condition $\textit{Info}(e)$ exige que e soit d'un type inductif informatif (ou bien logique vide ou singleton). On étend naturellement $\rightarrow_{\mathcal{E}}$ pour extraire les contextes.

Le non-déterminisme provient des deux règles $(\mathcal{E}\text{–}\square_1)$ et $(\mathcal{E}\text{–}\square_2)$. Le cas échéant, on peut en effet soit utiliser l'une de ces deux règles, soit utiliser une des règles structurelles. Ainsi, si une variable x a `Prop` pour sorte, on aura aussi bien $x \rightarrow_{\mathcal{E}} \square$ que $x \rightarrow_{\mathcal{E}} x$. Évidemment, la fonction $\mathcal{E}$ est une manière de déterminiser cette relation, en choisissant d'élaguer le plus tôt possible :

Lemme 7 *Si t est un terme* Cci *typable dans un contexte Γ, alors $t \rightarrow_{\mathcal{E}} \mathcal{E}(t)$.*

Preuve. Il suffit de toujours choisir les règles extrayant vers $\square$ dès que l'on a un schéma de type ou un terme logique. Il faut juste vérifier la condition auxiliaire *Info(e)* de la règle $(\mathcal{E}\text{–}case)$: Si un terme `case` entier n'est pas de sorte `Prop`, alors le terme e filtré est nécessairement d'un type inductif informatif (ou bien logique vide ou singleton). La condition *Info(e)* est donc bien vraie. □

Lemme 8 *Soit t un terme de* Cci *typable dans un contexte Γ, et t' dans* Cci$_\square$ *tel que $t \rightarrow_{\mathcal{E}} t'$. On a les propriétés immédiates suivantes :*

1. *t et t' diffèrent uniquement à des positions où t' contient des $\square$.*
2. *tout sous-terme de t correspondant à un $\square$ dans t est de sorte* `Prop` *ou bien un schéma de types.*
3. *tous les* `case` *restant dans t portent sur des types inductifs qui sont informatifs, singletons logiques ou vides.*

Cette relation $\rightarrow_{\mathcal{E}}$ présente l'avantage d'être stable par substitution, contrairement à la fonction $\mathcal{E}$:

Lemme 9 *Soient t,u,T,U quatre termes de* Cci *et* Γ *un contexte tels que :*

$$\begin{cases} \Gamma; (x : U) \vdash t : T \\ \Gamma \vdash u : U \end{cases}$$

Supposons également que l'on a t', u' *dans* $\text{Cci}_\Box$ *tels que* $t \to_\mathcal{E} t'$ *et* $u \to_\mathcal{E} u'$*. On a alors :*

$$t\{x \leftarrow u\} \to_\mathcal{E} t'\{x \leftarrow u'\}$$

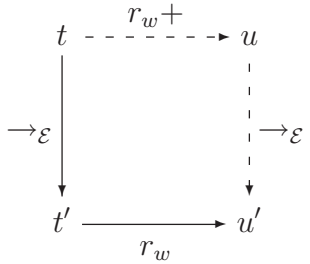
Preuve. Par récurrence sur la dérivation de $t \to_\mathcal{E} t'$, et par cas selon la dernière règle utilisée dans cette dérivation :

- ($\mathcal{E}$–$\Box_1$) ou ($\mathcal{E}$–$\Box_2$) : si t est un schéma de types ou de sorte `Prop`, alors il en est de même pour $t\{x \leftarrow u\}$ d'après le lemme de stabilité 1. On a alors bien $t\{x \leftarrow u\} \to_\mathcal{E} \Box$ par la même règle.
- ($\mathcal{E}$–*id*) : le cas où t n'est pas la variable x est évident, puisqu'il n'y a alors rien à substituer. Si au contraire $t = x = t'$, alors $x\{x \leftarrow u\} = u \to_\mathcal{E} u' = x\{x \leftarrow u'\}$.
- ($\mathcal{E}$–*lam*) : on a $t = \lambda y : Y,\ t_0$ et $t' = \lambda y : \Box,\ t'_0$ avec $t_0 \to_\mathcal{E} t'_0$. L'hypothèse de récurrence nous donne $t_0\{x \leftarrow u\} \to_\mathcal{E} t'_0\{x \leftarrow u'\}$. Or $(\lambda y : Y,\ t_0)\{x \leftarrow u\} = \lambda y : Y\{x \leftarrow u\},\ t_0\{x \leftarrow u\}$ et $(\lambda y : \Box,\ t'_0)\{x \leftarrow u'\} = \lambda y : \Box,\ t'_0\{x \leftarrow u'\}$. Ces deux derniers termes sont bien reliés par $\to_\mathcal{E}$, grâce à la règle($\mathcal{E}$–*lam*).
- ($\mathcal{E}$–*case*) : comme pour la règle ($\mathcal{E}$–*lam*), on obtient comme hypothèse de récurrence le passage à la substitution de chaque sous-expression du terme `case`. Avant d'utiliser la règle ($\mathcal{E}$–*case*) sur le `case` entier substitué, il faut juste s'assurer que la condition *`Info`*(e) reste vraie après substitution du terme e filtré dans le `case`. Or ceci est évident, car un terme inductif ne change pas son type inductif par substitution.
- les règles structurelles restantes se traitent comme ($\mathcal{E}$–*lam*).

□

Le théorème suivant exprime que l'on peut simuler au niveau `Coq` toute réduction faible d'un terme extrait.

Théorème 2 *Soient t un terme* Cci *clos bien typé et* t', u' *deux termes de* $\text{Cci}_\Box$ *tels que* $t \to_\mathcal{E} t'$ *et* $t' \to_{r_w} u'$*. Il existe alors un terme* Cci *u tel que* $u \to_\mathcal{E} u'$ *et* $t \to_{r_w+} u$*.*



Preuve. On procède par cas selon la réduction employée entre t' et u'. Nous commencerons par les deux cas délicats :

- La réduction effectuée est une ι_w-réduction singleton comme

  $\texttt{case}_n(\Box, \ldots, f') \to_\iota (f'\ \Box\ \ldots\ \Box)$.

Les règles de compatibilité pour la ι_w-réduction font que cette réduction se produit hors de tout lieur. Comme de plus on a interdit la présence d'axiomes, le sous-terme a de t qui correspond au $\Box$ éliminé est donc typable dans un contexte sans hypothèse. Comme a est un terme inductif, il peut alors être réduit vers un terme ayant un constructeur en tête : $(C\ \overrightarrow{p}\ \overrightarrow{v})$. Il est même possible d'effectuer cette réduction vers un constructeur de façon faible. En fait, C est le constructeur unique de ce type inductif singleton logique, et C a exactement n arguments en dehors des paramètres : $|\overrightarrow{v}| = n$. Donc dans t le sous-terme $\texttt{case}_n(a, \ldots, f)$ peut se réduire vers $(f\ v_1\ \ldots\ v_n)$ en au moins un pas de r_w-réduction. Il suffit de prendre maintenant pour u le terme issu de t par ces réductions. Pour vérifier que $u \to_{\mathcal{E}} u'$ nous devons seulement vérifier que $(f\ v_1\ \ldots\ v_n) \to_{\mathcal{E}} (f'\ \Box\ \ldots\ \Box)$. Ce qui est trivial, car tous les v_i sont en particulier de sorte `Prop` puisque arguments du constructeur d'un inductif singleton logique.

- La réduction effectuée est une ι_w-réduction d'un point-fixe dont l'argument « garde » dans t' est $\Box$. Alors l'argument « garde » correspondant g dans t possède un type inductif. En outre, comme dans le cas précédent, g est typable dans un contexte sans hypothèse, et peut donc être réduit vers un terme h débutant par un constructeur. On peut alors réduire le point-fixe dans t. Et finalement les termes ainsi obtenus dans Cci et dans $\text{Cci}_\Box$ sont toujours reliés par $\to_{\mathcal{E}}$.

Les autres cas sont bien plus aisés. Considérons par exemple le cas où la réduction effectuée est une β_w-réduction. Étant donnée la définition de la relation d'extraction, le β-redex dans t' possède forcément un β-redex correspondant dans t. Il suffit alors de réduire ce redex de t pour obtenir un u convenable. On a en effet bien $u \to_{\mathcal{E}} u'$, via le lemme précédent de substitution pour $\to_{\mathcal{E}}$. Enfin, tous les cas restants ($\to_{\delta_w}$, $\to_{\zeta_w}$ et la fin de $\to_{\iota_w}$) sont similaires à ce cas $\to_{\beta_w}$. □

Corollaire 1 *Soient t clos bien typé dans* Cci *et t' tels que $t \to_{\mathcal{E}} t'$. Alors toute suite de dérivations $\to_{r_w}$ partant de t' est finie.*

Preuve. Grâce à des applications répétées du théorème précédent, on peut en effet bâtir une suite correspondante de dérivations dans Cci partant de t, et au moins aussi longue. Or la forte normalisation de Cci affirme que cette dernière suite est finie. □

Cette terminaison est évidemment une bonne chose, mais ne suffit pas à assurer que la réduction faible d'un terme extrait se déroule sans accroc. En effet une terminaison prématurée, anormale, n'est pas non plus souhaitable. Pourrait-on terminer sur un terme normal qui ne soit pas une valeur, comme par exemple `(O true)` ou `match (fun x ⇒ x) with ...` ? Pour ces deux exemples grossiers, il est clair que la réponse est non : sinon le théorème précédent montrerait que ces deux termes pourraient être reliés par $\to_{\mathcal{E}}$ avec des termes bien typés de Cci, ce qui ici est impossible.

Par contre une forme normale tout à fait possible est l'application $(\Box\ \texttt{O})$. Il suffit pour cela de partir par exemple d'un prédicat $P : \forall n : \texttt{nat}, \texttt{True}$. On a alors bien $(P\ \texttt{O}) \to_{\mathcal{E}} (\Box\ \texttt{O})$, qui est normal vis-à-vis de r_w. Bien sûr, on a aussi $(P\ \texttt{O}) \to_{\mathcal{E}} \Box$, et c'est le choix que fait la fonction $\mathcal{E}$ initialement. Mais on a déjà vu avec l'exemple 1 que ce $(\Box\ \texttt{O})$ peut apparaître comme sous-terme en cours de réduction. Et avec une stratégie à la Ocaml qui demande en

premier l'évaluation des arguments, on se retrouve alors à chercher une valeur pour ($\Box$ `0`). Au niveau théorique, la réponse est la réduction ad hoc $\rightarrow_{\Box}$ déjà évoquée. Nous verrons dans les sections 2.6.3 et 3.3.2 comment réaliser cette réduction en pratique.

Lemme 10 *Dans* Cci$_{\Box}$*, toute suite de réductions* $\rightarrow_{\Box_w}$ *est finie.*

Preuve. Une réduction $\rightarrow_{\Box_w}$ fait décroître strictement la taille du terme. □

Lemme 11 *Soient* t *bien typé dans* Cci *et* t', t'' *tels que* $t \rightarrow_{\mathcal{E}} t'$ *et* $t' \rightarrow_{\Box_w} t''$*. On a alors* $t \rightarrow_{\mathcal{E}} t''$*.*

Preuve. Il suffit de considérer le redex ($\Box$ v') de t' que l'on réduit pour donner t''. Ce redex correspond à un sous-terme (u v) de t. On sait alors que u est soit un schéma de type, soit de sorte `Prop` (cf. lemme 8). Or ceci étant stable par application, il en est donc de même pour le sous-terme (v u). Il suffit alors d'appliquer ($\mathcal{E}-\Box_1$) ou ($\mathcal{E}-\Box_2$) un niveau plus tôt dans t. □

Théorème 3 *Soient* t *clos bien typé dans* Cci *et* t' *tels que* $t \rightarrow_{\mathcal{E}} t'$*. Alors toute suite de dérivations* $\rightarrow_{(r_w|\Box_w)}$ *partant de* t' *est finie.*

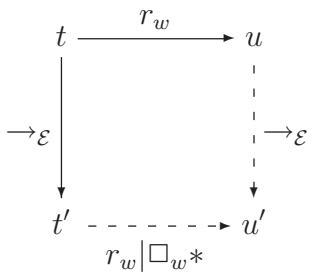
Preuve. Soit $t'_0 \dots t'_n \dots$ cette suite dans Cci$_{\Box}$. On bâtit une suite de Cci vérifiant à chaque étape $t_n \rightarrow_{\mathcal{E}} t'_n$:

- si $t'_n \rightarrow_{\Box_w} t'_{n+1}$, on prend $t_{n+1} = t_n$ et l'invariant correspond au lemme précédent.
- si $t'_n \rightarrow_{r_w} t'_{n+1}$, on utilise le théorème 2, et on obtient t_{n+1} tel que $t_n \rightarrow_{r_w} t_{n+1}$.

Tout d'abord, dans cette suite de Cci, il ne peut y avoir qu'un nombre fini d'étapes successives d'égalité, car elles correspondent à des réductions successives $\rightarrow_{\Box_w}$ au niveau Cci$_{\Box}$. Cette suite de Cci est donc constituée d'étapes de réductions entrecoupées éventuellement d'un nombre fini d'étapes d'égalité à chaque fois. Or à cause de la forte normalisation, il ne peut y avoir qu'un nombre fini de telles étapes de réduction. La suite dans Cci est donc finie, tout comme la suite d'origine. □

L'intégration de cette réduction $\rightarrow_{\Box_w}$ n'affecte donc pas la terminaison lors de l'évaluation d'un terme extrait. On peut maintenant se reposer la question de l'allure des formes normales vis-à-vis de la réduction $\rightarrow_{(r_w|\Box_w)}$. Pour répondre à cette question, nous allons avoir besoin d'un résultat dual du théorème 2 :

Théorème 4 *Soient* t, u *deux termes* Cci *bien typés (non nécessairement clos) et* t' *un terme de* Cci$_{\Box}$ *tels que* $t \rightarrow_{\mathcal{E}} t'$ *et* $t \rightarrow_{r_w} u$*. Il existe alors un terme* Cci$_{\Box}$ u' *tel que* $u \rightarrow_{\mathcal{E}} u'$ *et vérifiant* $t' \rightarrow_{r_w} u'$ *ou bien* $t' \rightarrow^*_{\Box_w} u'$*.*



PREUVE.

- Si le redex r réduit dans t correspond à un redex similaire dans t' qui est complet, alors il suffit de réduire ce redex de t' pour obtenir un u' qui convienne.
- Si r est complètement contenu dans un sous-terme de t correspondant à un $\Box$ de t', il suffit alors de prendre $u' = t'$.
- Nous allons maintenant considérer les cas intermédiaires où r correspond à un redex de t' incomplet car en partie absorbé par $\Box$. Ces positions correspondent aux cas 1, 2, 3a et 3b de la section 2.3.1.
  - Si r est un β-redex, la seule situation à considérer est $r = (\lambda x : X,\, a)\; b$ dans t correspondant à $(\Box\; b')$ dans t'. On peut alors simuler la β-réduction de t par une $\Box$-réduction dans t'.
  - Si r est un δ- ou ζ-redex, il n'y pas de situation intermédiaire.
  - Si r est un ι-redex `case`, le seul cas restant est

    $\texttt{case}(e,\, P,\, \ldots)$

    dans t correspondant dans t' à

    $\texttt{case}(\Box,\, P',\, \ldots)$.

    Les propriétés de $\rightarrow_{\mathcal{E}}$ (lemme 8) font que e est soit de sorte `Prop` soit un schéma de types. Comme e est un terme inductif, qui ne peut donc être un schéma de types, e est finalement de sorte `Prop`. D'autre part, la condition ***`Info`*** nous apprend que ce `case` dans t porte sur un inductif informatif, ou bien vide, ou bien singleton logique. Compte tenu de la sorte de e, le cas informatif est impossible. Il ne s'agit pas non plus d'un inductif vide, qui ne pourrait pas se réduire faute de constructeur. Nous sommes donc en présence d'une élimination singleton logique, qu'il est maintenant possible de réduire grâce à la nouvelle ι-réduction.
  - Si r est un ι-redex `fix`, il y a alors deux sous-cas. Si le `fix` a disparu dans t' mais pas tous les arguments composant le redex initial (cas 3a), alors on peut simuler la ι-réduction de t par quelques $\Box$-réductions. Et si le `fix` est présent dans t', c'est alors l'argument « de garde » qui est $\Box$ (cas 3b). On réduit alors grâce à la nouvelle ι-réduction des `fix`.

□

Théorème 5 *Soient t un terme clos de* CCI *bien-typé et t' dans* $\text{CCI}_{\Box}$ *tels que $t \rightarrow_{\mathcal{E}} t'$. Alors toute forme normale t'_0 de t' modulo $\rightarrow_{(r_w|\Box_w)}$ correspond via $\rightarrow_{\mathcal{E}}$ à une forme normale faible t_0 de t. Plus précisément, on est dans l'un des quatre cas suivants :*

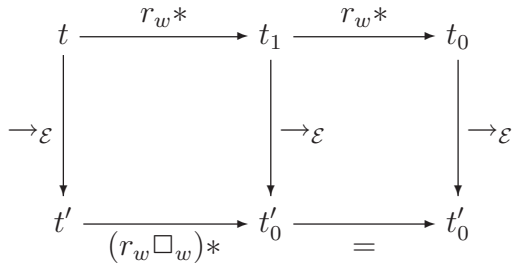
(i) $t'_0 = \Box$

(ii) $t'_0 = C\; \overrightarrow{v}$ *et les arguments $\overrightarrow{v}$ sont alors aussi des formes normales modulo $\rightarrow_{(r_w|\Box_w)}$.*

(iii) t'_0 *commence par une λ-abstraction.*

(iv) $t'_0 = \texttt{fix}\; f_i\; \{\ldots\}\; \overrightarrow{v}$. *Si l'argument de garde est le k_i-ème, alors $|\overrightarrow{v}| < k_i$ et les $\overrightarrow{v}$ sont également en forme normales modulo $\rightarrow_{(r_w|\Box_w)}$.*

Preuve. Comme lors de la preuve du théorème 3, on construit une suite de dérivations partant de t et reflétant au niveau Cci la suite de dérivations menant de t' à t'_0. Ceci nous donne un terme Cci t_1 tel que $t \to^*_{r_w} t_1$. Si maintenant on continue à appliquer à t_1 des réductions faibles, on finit par obtenir un terme t_0 de Cci faiblement normal. On peut alors refléter cette dérivation $t_1 \to^*_{r_w} t_0$ au niveau Cci$_\Box$ via le théorème 4, ce qui nous donne une suite de dérivations $\to_{(r_w \Box_w)*}$ partant de t'_0. Or ce dernier est en forme normale vis-à-vis de ces réductions, donc ne change pas. Finalement, on a le schéma suivant :



Le reste de l'énoncé découle directement de l'étude des allures possibles d'une forme normale faible close dans Cci (cf. lemme 3). En effet t_0 peut être :

- une sorte, un type inductif appliqué ou un produit, qui ne peuvent que donner $\Box$ d'après les règles de $\to_\mathcal{E}$.
- un constructeur inductif appliqué, qui donne par $\to_\mathcal{E}$ soit $\Box$ soit le même constructeur appliqué à des arguments extraits.
- une λ-abstraction, qui donne par $\to_\mathcal{E}$ soit $\Box$ soit une λ-abstraction dérivée.
- un point-fixe manquant d'arguments, qui donne par $\to_\mathcal{E}$ soit $\Box$ soit un point-fixe dérivé.

$\Box$

Même s'il ne mentionne que la réduction générale $\to_{r_w}$, ce théorème est également intéressant du point de vue des stratégies particulières d'évaluation, stricte (à la Ocaml) ou paresseuse (à la Haskell). En effet, on a déjà mentionné précédemment que ces deux stratégies peuvent être vues comme des restrictions de $\to_{r_w}$, avec comme condition supplémentaire que toute partie droite (resp. gauche) d'application soit en forme normale avant de pouvoir réduire l'autre côté. Dans tous les cas, le terme retourné à la fin d'une évaluation stricte est clairement normal vis-à-vis de $\to_{r_w}$, et de même pour une évaluation paresseuse. Pour pouvoir utiliser le théorème précédent, il faut juste que ces évaluations strictes ou paresseuses intègrent également la réduction $\to_{\Box_w}$. Dans la section 2.6, nous verrons comment concilier nos besoins en terme de règles de réduction avec les évaluateurs réellement implantés dans Ocaml et Haskell.

Il est possible de donner plus de précisions concernant les quatre cas du théorème précédent. Si $t'_0 = \Box$ alors t_0 est un schéma de type ou est de sorte `Prop`. Le lemme 2 montre qu'il en est alors de même pour t. Dit autrement (par contraposé) : si t est informatif et n'est pas un schéma de types, alors la réduction de son pendant extrait ne peut terminer sur $\Box$.

Réciproquement, on aimerait que $t'_0 = \Box$ dès que t est logique ou est un schéma de types. Cela est inexact, à cause du non-déterminisme de $\to_\mathcal{E}$, comme le montre l'exemple

$\lambda x : \texttt{True}$, $x \to_{\mathcal{E}} \lambda x : \Box, \Box$. Mais bien sûr, il suffit de spécialiser le théorème précédent avec $t' = \mathcal{E}(t)$ au lieu de $t \to_{\mathcal{E}} t'$ pour assurer cette propriété. De toute manière, $\to_{\mathcal{E}}$ n'a d'utilité que comme invariant intermédiaire.

Concernant les autres cas du résultat :

- Pour (ii) : si t'_0 commence par un constructeur, alors t a soit un type inductif si le constructeur est complètement appliqué, soit un type produit sinon.
- Réciproquement : si $t' = \mathcal{E}(t)$, et si t possède un type inductif informatif, alors on termine nécessairement la réduction dans le cas (ii).
- Si l'on est dans le cas (iii) ou le cas (iv), c'est que t avait initialement un type produit.
- Par contre si l'on sait juste que t a type produit, on peut finir dans n'importe quel cas (i) (ii) (iii) ou (iv).

Le bilan de cette étude est que la réduction des termes extraits se déroule sans anicroches. Si la machine à réduction doit effectuer une application, elle va bien trouver en tête une valeur acceptant un argument : clôture, point-fixe ou $\Box$ (ou un constructeur inductif tant qu'on garde leur notation curryfiée). Et si la machine à réduction doit effectuer un filtrage, l'objet filtré va bien se réduire vers un constructeur appliqué à tous ses arguments, hormis les cas spéciaux des filtrages de singletons logiques.

La question est maintenant de savoir si le résultat final de la réduction d'un terme extrait est bien correcte vis-à-vis du terme initial. Évidemment, le travail fait jusqu'ici va permettre de donner une première réponse, au moins dans les cas simples, comme par exemple le cas particulier des termes appartenant à un type de données :

Définition 15 *Un type de données est un type inductif D dont les constructeurs n'ont pour arguments que des objets de type D ou d'un autre type de données.*

Par exemple, un type inductif I avec un constructeur de type $(\texttt{nat} \to I) \to I$ (*i.e.* encapsulant une fonction) n'est pas un type de données. Par contre les types usuels comme `bool`, `nat` ou `Z` sont des types de données en ce sens. On peut alors spécialiser notre résultat précédent à ce cas particulier :

Théorème 6 *Soit t un terme clos de* Cci *bien-typé dont le type T est un type de données sans contenu logique. Alors toutes les dérivations de $\mathcal{E}(t)$ via $\to^*_{r_w \Box_w}$ se terminent sur la forme normale* Cci *de t.*

Preuve. Il suffit de reprendre la preuve du théorème précédent, à ceci près que maintenant t_0 est en forme normale et non plus seulement en forme normale faible. Ceci découle du fait que T est un type de données : une forme normale faible dans ce type, ne pouvant contenir de clôtures, est donc également en forme normale. Et dès lors, la définition de sans contenu logique montre que $\mathcal{E}(t_0) = t_0$. Il n'y a donc rien à extraire dans t_0, on a donc forcément $t'_0 = t_0$. □

Par exemple, si l'on bâtit en Coq un terme arbitrairement complexe répondant `true` ou `false` à une question particulière, on est sûr que son extraction va se réduire vers la même réponse que celle de Coq. Il en va de même si notre terme retourne le centième nombre de Fibonacci ou bien la liste des mille premières décimales de π.

Ce résultat, bien qu'intéressant, est malgré tout fortement limité. On ne dit rien en particulier concernant la correction de termes non clos ou de fonctions (ce qui en fait revient au même, via des λ-abstractions). Pour traiter ces cas, il va falloir tout d'abord préciser ce qu'on entend par fonctions correctes, puis établir cette correction. L'idée simple est de montrer que si tous les arguments sont corrects en un certain sens, alors la sortie calculée par la fonction l'est également. La formalisation de cette idée et sa preuve se sont révélées beaucoup plus ardues que prévu, et sont l'objet de toute la section suivante.

2.4 Étude sémantique de la correction de l'extraction

Dans l'étude qui suit, nous allons nous efforcer de préciser ce que cela signifie pour un terme extrait d'être correct, et nous allons désormais le faire au moyen de considérations sémantiques. Jusqu'ici nous avons en effet utilisé une propriété de correction purement syntaxique, à savoir la comparaison entre la structure d'un terme extrait et celle du terme d'origine. Mais une telle approche ne permet pas de traiter les fonctions de façon satisfaisantes.

Cette étude est destinée à permettre à terme de générer dans Coq les preuves de correction des objets extraits. Cette intention a influé grandement sur l'étude, à la fois dans le choix du cadre logique, puis dans les définitions et les preuves qui suivent, faites de la manière la plus détaillée et mécanique possible.

Nous allons tout d'abord définir une transformation $[\![.]\!]$, qui va en particulier nous donner, une fois appliqué à un type T, le prédicat de correction que doit vérifier tout terme $\mathcal{E}(t)$ extrait à partir d'un objet t de type T. Puis nous établirons successivement :

- la préservation de cette transformation $[\![.]\!]$ par substitution ;
- la préservation de $[\![.]\!]$ par réduction ;
- le fait que les objets construits par $[\![.]\!]$ sont correctement typés ;
- le fait que les termes extraits vérifient bien les prédicats de correction issus de $[\![.]\!]$.

Avertissement : Pour tenter de simplifier cette étude, nous ne traitons pas les cas des « let-in », des constantes et des points-fixes à plus d'une composante. Il ne s'agit de toute façon pas des cas critiques, et leur prise en compte, fastidieuse, ne pose a priori pas de soucis. Par contre le principal changement dans cette section est l'utilisation du système modifié CCI_m avec marquage explicite des cumulativités, introduit dans la section 1.2.5.

2.4.1 Cadre logique

On se place ici dans une perspective aussi proche que possible d'une formalisation réelle en Coq de la correction de l'extraction. Même si cette formalisation est restée au cours de cette thèse dans le cadre strict du papier, il est possible que nous tentions à l'avenir d'en faire un véritable développement Coq. Le système logique dans lequel nous allons exprimer nos propriétés de correction est ici le CCI_m.

Nous avons déjà évoqué à la page 10 de l'introduction le fait qu'il n'est pas possible en général d'espérer pouvoir typer les termes extraits dans le CCI_m, à cause par exemple de la

disparition éventuelle des certificats logiques de décroissance dans un point-fixe. Il va donc falloir encapsuler ces termes extraits dans un type de données, du genre :

```
Inductive expr : Set :=
 | Var : identifier → expr
 | Lam : identifier → expr → expr
 | App : expr → expr → expr
 | ...
```

Nous supposons désormais que l'on dispose dans notre Cci_m d'un tel type concret, que l'on nommera Λ par la suite, internalisant la syntaxe des termes non-typés du Cci_m (ou pré-termes), plus une constante $\Box$. De façon générale, on notera t_Λ l'objet t internalisé dans Λ. Par abus de notation, on continuera à utiliser la syntaxe Cci_m pour ces objets internalisés. Ainsi on écrira ($\mathtt{S}_\Lambda$ $\mathtt{O}_\Lambda$) plutôt que ($\mathtt{App}$ $\mathtt{S}_\Lambda$ $\mathtt{O}_\Lambda$).

De même que deux objets de Cci_m convertibles ont les mêmes propriétés, et en particulier sont égaux via l'égalité standard `eq`, nous allons exiger que deux objets Λ convertibles soient égaux. En particulier, nous utiliserons le fait que `((λx:X, t) x)` et `t` sont le même objet. Nous n'allons pas, pour l'instant, préciser plus quelle notion de convertibilité est nécessaire au niveau Λ, quitte à le faire au fur et à mesure des besoins.

Ce cadre logique étant fixé, la fonction $\mathcal{E}$ d'extraction vue au début du chapitre est maintenant une fonction au niveau méta, transformant tout terme Cci_m en un objet du type concret Λ.

2.4.2 Les prédicats de simulation

Dans les travaux similaires de correction d'extraction, l'usage est de définir la correction des termes extraits par rapport au type initial de l'objet extrait. On a alors un prédicat de réalisabilité p `r` T, qui se lit ainsi : « le programme p réalise le type T ». Le but est alors de montrer que $\mathcal{E}(t)$ `r` T lorsque $t : T$. Le point critique est la réalisation des fonctions. La règle naturelle pour réaliser un type fonctionnel est la suivante :

$$p \ \mathtt{r} \ A \to B \ \text{ ssi } \ \forall a,\ a \ \mathtt{r} \ A \Rightarrow (p\ a) \ \mathtt{r} \ B$$

Si l'on veut généraliser ceci au produit dépendant $\forall x : A,\ B$, il faut tenir compte de l'apparition possible de x dans B, que l'on va souligner par la notation $B(x)$. Essayons :

$$p \ \mathtt{r} \ \forall \mathrm{x} : \mathrm{A}, \mathrm{B(x)} \ \text{ ssi } \ \forall a,\ a \ \mathtt{r} \ A \Rightarrow (p\ a) \ \mathtt{r} \ B(a)$$

Dans un système permettant une extraction interne, ceci peut éventuellement convenir, encore que a `r` A n'implique pas forcément $a : A$, et donc $B(a)$ peut ne pas être bien typé. Ici en tout cas, comme les parties extraites ont pour type Λ, il n'y a aucune chance pour que cette règle soit bien typée. On peut alors tenter de s'appuyer sur un élément x de type A :

$$p \ \mathtt{r} \ \forall \mathrm{x} : \mathrm{A}, \mathrm{B(x)} \ \text{ ssi } \ \forall x : A,\ \forall a,\ a \ \mathtt{r} \ A \Rightarrow (p\ a) \ \mathtt{r} \ B(x)$$

Dans cette formulation, le point gênant maintenant est que x et a ne sont pas corrélés, bien que moralement ils doivent correspondre respectivement à un terme Cci_m et à une extraction

possible de ce terme. Pour exprimer cette corrélation, nous avons choisi d'introduire un prédicat de simulation $t \sim p$ reliant un terme $\text{CCI}_m\ t$ et un terme extrait p. Plus exactement, il va s'agir d'une famille de prédicats $\sim_T$ indexés par des types T de CCI_m :

$$\sim_T\ :\ T \to \Lambda \to \texttt{Prop}$$

La règle de réalisation d'un produit devient alors :

$$f \sim_{\forall x:A,\, B(x)} p \quad \text{ssi} \quad \forall x{:}A,\ \forall a,\ x \sim_A a\ \Rightarrow\ (f\ x) \sim_{B(x)} (p\ a)$$

En fait, nous allons tout centrer sur ces prédicats $\sim_T$, et reléguer à l'arrière-plan le prédicat de réalisabilité `r`, qui sera en fait défini à partir des prédicats de simulation :

$$p\ \texttt{r}\ T \ \text{ ssi } \ \exists t{:}T,\ t \sim_T p$$

Et pour atteindre l'objectif initial qui était d'établir $\mathcal{E}(t)$ `r` T, on va alors prouver que $t \sim_T \mathcal{E}(t)$, ce qui est plus précis.

Au niveau technique, ces prédicats $\sim_T$ pour $T : s$ ne sont encore pas assez généraux pour être définis et manipulés directement. Nous allons en effet avoir besoin de les étendre aux schémas de types $T : K$. Comme en général, ces prédicats ne seront plus des relations binaires, nous ne garderons pas cette notation infixe $\sim$, et parlerons à la place des prédicats $\widehat{T}$. Lorsque T est un type, $t \sim_T p$ sera alors juste une abréviation pour $(\widehat{T}\ t\ p)$.

En pratique, nous n'allons pas pouvoir définir directement ces prédicats $\widehat{T}$, mais plutôt des paires dépendantes $[\![T]\!]$ qui vont avoir ces prédicats $\widehat{T}$ comme seconde composante, et une variante enrichie $\overline{T}$ de T comme première composante. Nous utiliserons donc trois types de paires dépendantes adaptées à nos besoins :

```
Record Type+ : Type := mk_Type
 { type_Type :> Type;
   pred_Type : type_Type → Λ → Prop }.

Record Set+ : Type := mk_Set
 { type_Set :> Set;
   pred_Set : type_Set → Λ → Prop }.

Record Prop+ : Type := mk_Prop
 { type_Prop :> Prop;
   pred_Prop : type_Prop → Λ → Prop := fun _ _ ⇒ True }.
```

Si s est une des trois sortes `Set`, `Prop` ou `Type`, le constructeur d'une paire dépendante de type s^+ est `mk_`s, et les deux projections sont `type_`s et `pred_`s. Ces deux projections redonnent respectivement le type contenu dans s^+ et le prédicat de simulation associé à ce type. Regardons maintenant les types des projections. Le type de la première est très simple :

```
type_s : s+ → s.
```

Celui de la deuxième présente une dépendance :

```
pred_s : ∀T:s+, (type_s T) → Λ → Prop.
```

D'autre part, la première projection peut être vue comme une coercion de s^+ vers s, ce qui est signalé par la syntaxe `:>` au lieu des `:` habituels. En Coq, cette coercion évite alors d'avoir à écrire explicitement la première projection. Dans l'étude théorique qui suit, nous continuerons à expliciter ces projections. Par contre, nous le ferons à l'aide d'une syntaxe plus légère :

- $T.\mathbb{1}$ pour (`type_`s T)
- $T.2$ pour (`pred_`s T)

Ces abréviations sont volontairement ambiguës, car elles ne précisent pas le type d'origine `Type`$^+$, `Set`$^+$ ou `Prop`$^+$. Typer l'expression T permet au besoin de lever cette ambiguïté.

Il faut noter que le cas de `Prop`$^+$ est un peu particulier. En effet on fixe définitivement le contenu du champ `pred_Prop` dès la définition du type `Prop`$^+$. Ainsi le prédicat associé à un objet dans `Prop`$^+$ est nécessairement le prédicat trivial toujours équivalent à `True`. L'idée est que l'extraction d'une partie logique peut être choisie arbitrairement, sans que cela ait de répercussion sur la correction du terme extrait. Au niveau pratique, la seule différence entre le type `Prop`$^+$et les types `Type`$^+$et `Set`$^+$est que le constructeur `mk_Prop` n'attend qu'un seul argument au lieu de deux. Par contre les deux projections `type_Prop` et `pred_Prop` existent bien et fonctionnent comme décrit précédemment.

L'homogénéité entre les s^+ va nous permettre de plonger `Set`$^+$ et `Prop`$^+$ dans `Type`$^+$ pour mimer les cumulativité s `Set` $<$ `Type` et `Prop` $<$ `Type`. Dans Cci$_m$, ces cumulativité s sont signalées par les marques ‡ et †. Ces marques vont ici être reflétées par les deux fonctions suivantes :

```
Definition Set+_Type+ := fun T:Set+ ⇒ let (t,p):=T in mk_Type t‡ p.
Definition Prop+_Type+ := fun T:Prop+ ⇒ let (t,p):=T in mk_Type t† p.
```

2.4.3 La transformation $[\![.]\!]$

Nous allons maintenant définir une transformation $[\![.]\!]$ pour tout objet de Cci$_m$. En fait, seule la transformation des types nous importe réellement. Mais ces types peuvent se retrouver à tout emplacement, par exemple en argument d'un constructeur inductif, et resurgir alors à la faveur d'un filtrage. Et un type peut aussi être en fait une application d'un schéma de types à des arguments. Bref, $[\![.]\!]$ va porter sur tout terme Cci$_m$.

Dans le cas particulier d'un type $T : s$, $[\![T]\!]$ va être une paire dépendante dont la deuxième composante $[\![T]\!].2$ est le prédicat de simulation recherché. Dans ce cas précis, nous abrégerons alors $[\![T]\!].\mathbb{1}$ en $[\![T]\!]_{\mathbb{1}}$ et $[\![T]\!].2$ en $[\![T]\!]_2$. Comme point de repère, on pourra noter que cette transformation méta-théorique $[\![.]\!]$ va préserver le typage : si $t : T$, alors $[\![t]\!] : [\![T]\!]_{\mathbb{1}}$. Voici maintenant la définition de $[\![.]\!]$ par récurrence structurelle :

– $[\![x]\!] = x$

– $[\![t\ t']\!] = [\![t]\!]\ [\![t']\!]$

- $[\![\lambda x : T,\, t]\!] = \lambda x : [\![T]\!]_{\mathbb{1}},\, [\![t]\!]$
- $[\![\forall x : T,\, T']\!] =$
  $\texttt{mk\_}s\ \ \forall x : [\![T]\!]_{\mathbb{1}},\, [\![T']\!]_{\mathbb{1}} \quad \lambda t,\, \lambda p,\, \forall x : [\![T]\!]_{\mathbb{1}},\, \forall x' : \Lambda,\, [\![T]\!]_2\ x\ x' \to [\![T']\!]_2\ (t\ x)\ (p\ x')$
  si le type[3] de $\forall x : T,\, T'$ est $s \neq \texttt{Prop}$. Les deux abstractions abrégées ont pour types $\forall x : [\![T]\!]_{\mathbb{1}},\, [\![T']\!]_{\mathbb{1}}$ et Λ. Et si $s = \texttt{Prop}$, on supprime le second argument de $\texttt{mk\_}s$.
- $[\![s]\!] = \ \texttt{mk\_Type}\ \ s^+\ \ \lambda\_,\, \lambda\_,\, \texttt{True}$
- $[\![t^{\ddagger}]\!] = \texttt{Set}^+\texttt{\_Type}^+\ [\![t]\!]$
- $[\![t^{\dagger}]\!] = \texttt{Prop}^+\texttt{\_Type}^+\ [\![t]\!]$
- $[\![I]\!] = \lambda \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\, (\texttt{mk\_}s\ (\overline{I}\ \overrightarrow{u})\ (\widehat{I}\ \overrightarrow{u}))$
  lorsque l'inductif I admet $\forall \overrightarrow{u : U},\, s$ pour arité, avec $s \neq \texttt{Prop}$. Voir la transformation des contextes pour la définition de $\overline{I}$ et $\widehat{I}$. Et si s est $\texttt{Prop}$, il suffit d'enlever le second argument de $\texttt{mk\_}s$.
- $[\![C]\!] = \overline{C}$ pour un constructeur C de l'inductif I. Et $\overline{C}$ est alors un constructeur de l'inductif $\overline{I}$. Voir la transformation des contextes plus bas pour la définition de $\overline{I}$.
- $[\![\texttt{case}(e,\, P,\, \overrightarrow{f_i})]\!] = \texttt{case}([\![e]\!],\, \overline{P},\, \overrightarrow{[\![f_i]\!]})$
  Ici, pour P de la forme $\lambda \overrightarrow{u},\, \lambda x : (I\ \overrightarrow{q}\ \ \overrightarrow{u}),\, T$, on note $\overline{P} = \lambda \overrightarrow{u},\, \lambda x : (\overline{I}\ \overrightarrow{[\![q]\!]}\ \ \overrightarrow{u}),\, [\![T]\!]_{\mathbb{1}}$.
- $[\![\texttt{fix}\ x : T := t]\!] = \texttt{fix}\ x : [\![T]\!]_{\mathbb{1}} := [\![t]\!]$

Finalement cette transformation $[\![.]\!]$ s'étend aux contextes comme suit :

- $[\![\Gamma; (x : T)]\!] = [\![\Gamma]\!]; (x : [\![T]\!]_{\mathbb{1}})$
- Chaque déclaration d'un inductif I d'arité $K = \forall \overrightarrow{u : U},\, s$ et de constructeurs $C_i : T_i$ est remplacée par deux inductifs $\overline{I}$ et $\widehat{I}$, le second ne servant que si $s \neq \texttt{Prop}$.
  1. $\overline{I}$ est simplement la propagation de $[\![.]\!]_{\mathbb{1}}$ à I. Son arité est $\overline{K} = \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\, s$ et ses constructeurs $\overline{C}_i$ ont pour type $[\![T_i]\!]_{\mathbb{1}}$.
  2. $\widehat{I}$ sert de prédicat de simulation pour $\overline{I}$. Il a pour arité $\widehat{K} = \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\, (\overline{I}\ \overrightarrow{u}) \to \Lambda \to \texttt{Prop}$. Et ses constructeurs $\widehat{C}_i$ ont pour type $([\![T_i]\!]_2\ \overline{C}_i\ C^i_\Lambda)$, avec C^i_Λ étant le i-ème constructeur de l'inductif $\mathcal{E}(I)$ extrait de I et internalisé dans Λ.

Revenons un instant sur les définitions de $\overline{I}$ et $\widehat{I}$. Il peut sembler en effet surprenant que $[\![T_i]\!]_{\mathbb{1}}$ d'une part et $([\![T_i]\!]_2\ \overline{C}_i\ C^i_\Lambda)$ d'autre part soient bien des types de constructeurs valides. En fait si T_i est de la forme $\forall \overrightarrow{v : V},\, (I\ \overrightarrow{w})$, on a alors :

$[\![T_i]\!]_{\mathbb{1}} = \forall \overrightarrow{v : [\![V]\!]_{\mathbb{1}}},\, (\overline{I}\ \overrightarrow{[\![w]\!]})$

et également :

$([\![T_i]\!]_2\ \overline{C}_i\ C^i_\Lambda) = \forall \overrightarrow{[\![v : V]\!]},\, (\widehat{I}\ \overrightarrow{S(w)}\ (\overline{C}_i\ \overrightarrow{v})\ (C^i_\Lambda\ \overrightarrow{v^\Lambda}))$

Nous avons ici utilisé deux nouvelles notations :

[3] Rappel : dans CCI_m, on a bien unicité des types modulo conversion.

- $\forall[\![\overrightarrow{x : T}]\!]$, ... qui dénote $\forall\overrightarrow{x : [\![T]\!]_{\mathbb{1}}}$, ... plus une variable de programme x' associée à chaque variable x de départ, et la preuve H_x de correction reliant x et x'. Plus précisément :

  $$\forall[\![(x : T)\overrightarrow{(v_i : V_i)}]\!], \ldots = \forall x : [\![T]\!]_{\mathbb{1}}, \forall x' : \Lambda, \forall H_x : ([\![T]\!]_2\ x\ x'), \forall[\![\overrightarrow{v_i : V_i}]\!], \ldots$$

- $\overrightarrow{u}^{\Lambda}$ est la version extraite des variables : chaque variable x est remplacée par sa variable extraite associée x'.

Pour un contexte Γ, on aura plus tard besoin de manipuler une variante de $[\![\Gamma]\!]$ encore plus enrichie, dans laquelle une déclaration $(x : T)$ engendre, en plus de $(x : [\![T]\!]_{\mathbb{1}})$, la déclaration d'une variable de programme associée $(x' : \Lambda)$ et d'une preuve H_x reliant x et x', de type $([\![T]\!]_2\ x\ x')$. Nous noterons ce contexte enrichi $[\![\Gamma]\!]_+$.

2.4.4 Un exemple

Il est maintenant temps d'essayer tout ce joli formalisme sur un exemple. Nous allons chercher à savoir quelle condition de correction doit satisfaire un programme extrait $\mathcal{E}(\texttt{div})$ dont le terme original `div` était une division entière :

```
div : ∀a b:nat, b≠0 → { q:nat | q*b ≤ a ∧ a < (S q)*b }
```

Tout d'abord, comme l'arité de `nat` est directement `Set`, on a $[\![\texttt{nat}]\!] = \texttt{mk\_Set}\ \overline{\texttt{nat}}\ \widehat{\texttt{nat}}$, et donc $[\![\texttt{nat}]\!]_{\mathbb{1}} = \overline{\texttt{nat}}$. Les types des nouveaux constructeurs $\overline{\texttt{O}}$ et $\overline{\texttt{S}}$ de $\overline{\texttt{nat}}$ sont alors respectivement $\overline{\texttt{nat}}$ et $\overline{\texttt{nat}} \to \overline{\texttt{nat}}$, ce qui fait que $\overline{\texttt{nat}}$ est exactement isomorphe à `nat`. Nous les identifierons donc. Et pour ce qui est de $\widehat{\texttt{nat}}$, on a :

```
Inductive nat̂ : nat → Λ → Prop :=
 | Ô : nat̂ O O_Λ
 | Ŝ : ∀n:nat, ∀n':Λ, nat̂ n n' → nat̂ (S n) (S_Λ n').
```

Cet inductif exprime simplement le fait que `n':Λ` est une extraction correcte de `(S (S ... (S O)...))` ssi `n' = (S_Λ (S_Λ ... (S_Λ O_Λ)...))`. Un autre inductif utilisé dans le type de `div` est `sig` :

```
Inductive sig̅ (A:Set⁺)(P:A.𝟙 → Prop⁺) : Set :=
 | exist̅ : ∀x:A.𝟙, (P x).𝟙 → sig̅ A P.
```

Cette définition n'est, en fait, pas si différente de celle de `sig`, en particulier si l'on omet les coercions `.𝟙` inférables par Coq. On a même la relation suivante :

$\overline{\texttt{sig}}\ A\ P \leftrightarrow \texttt{sig}\ \ A.\mathbb{1}\ \ \lambda x, ((P\ x).\mathbb{1})$

Voici maintenant la définition de l'inductif $\widehat{\texttt{sig}}$:

```
Inductive sig̀ (A:Set⁺)(P:A.𝟙 → Prop⁺) : sig̅ A P → Λ → Prop :=
 | exist̀ : ∀x:A.𝟙, ∀x':Λ, A.2 x x' →
    ∀h:(P x).𝟙, ∀h':Λ, (P x).2 h h' →
```

.../...

```
                                    .../...
     sig A P (exist x h) (exist_Λ x' h')).
```

Nommons `Div` le type de `div` et `P` le prédicat `(fun a b q ⇒ q*b ≤ a ∧ a < (S q)*b)`. On obtient ensuite que $[\![\texttt{Div}]\!]_1$ est isomorphe à `Div` puis que $[\![\texttt{Div}]\!]_2$ a pour type `Div` → Λ → `Prop` et vaut :

```
[[Div]]_2 = fun (f:Div)(p:Λ) ⇒
 ∀a:nat, ∀a':Λ, nat a a' →
 ∀b:nat, ∀b':Λ, nat b b' →
 ∀h:b≠0, ∀h':Λ, [[b≠0]]_2 h h' →
 (sig [[nat]] ([[P]] a b) (f a b h) (p a' b' h')).
```

Or `b≠0` étant une proposition logique, le prédicat $[\![\texttt{b≠0}]\!]_2$ qui lui est associé est le prédicat trivial. Et de même pour tout triplet d'entier `a`, `b` et `q`, le prédicat ($[\![\texttt{P}]\!]$ *a b q*).2 est aussi λ_, λ_, `True`. Au final, $[\![\texttt{Div}]\!]_2$ est équivalent à :

```
fun (f:Div)(p:Λ) ⇒
 ∀a:nat, ∀a':Λ, nat a a' →
 ∀b:nat, ∀b':Λ, nat b b' →
 ∀h:b≠0, ∀h':Λ,
 ∃q, ∃q', (f a b h) = (exist q _) ∧ (p a' b' h') = (exist_Λ q' _) ∧
         nat q q' ∧ P a b q.
```

Notre fonction extraite `div` sera donc correcte ssi pour deux arguments correspondant à des entiers Coq (le deuxième étant non-nul) et un troisième argument quelconque, elle se réduit bien vers le constructeur $\texttt{exist}_\Lambda$ dont le premier argument voit son pendant Coq vérifier la post-condition `P`. Malgré tous les détours liés à la complexité du formalisme, on retrouve bien la signification informelle des pré- et post-conditions.

2.4.5 Propriétés de substitution liées à la transformation $[\![.]\!]$

La première étape en direction de la preuve de correction de l'extraction est d'établir les propriétés de substitution que vérifie la transformation $[\![.]\!]$. Il va s'agir ici de substitution de la dernière variable d'un contexte, ce qui en particulier n'influe pas sur les types inductifs définis auparavant.

Lemme 12 *La transformation* $[\![.]\!]$ *préserve les substitutions :* $[\![t\{x \leftarrow r\}]\!] = [\![t]\!]\{x \leftarrow [\![r]\!]\}$

Preuve. Il s'agit d'une preuve par récurrence structurelle, purement syntaxique. Devant la multitude de cas, nous ne les traiterons pas tous. Voici un cas typique :

$$\begin{aligned}
[\![(t\ t')\{x \leftarrow r\}]\!] &\overset{subst.}{=} [\![t\{x \leftarrow r\}\ t'\{x \leftarrow r\}]\!] \\
&\overset{def.\ [\![.]\!]}{=} [\![t\{x \leftarrow r\}]\!]\ [\![t'\{x \leftarrow r\}]\!] \\
&\overset{hyp.\ rec.}{=} [\![t]\!]\{x \leftarrow [\![r]\!]\}\ [\![t']\!]\{x \leftarrow [\![r]\!]\} \\
&\overset{subst.}{=} ([\![t]\!]\ [\![t']\!])\{x \leftarrow [\![r]\!]\} \\
&\overset{def.\ [\![.]\!]}{=} [\![t\ t']\!]\{x \leftarrow [\![r]\!]\}
\end{aligned}$$

Tous les cas autres que les cas de base suivent ce schéma :

1. propagation de la substitution
2. utilisation de la définition de $[\![.]\!]$.
3. utilisations répétées des hypothèses de récurrence.
4. factorisation de la substitution
5. utilisation inversée de la définition de $[\![.]\!]$.

Il y a quand même un point délicat : rien ne garantit a priori que la même règle de définition de $[\![.]\!]$ va s'appliquer avant (point 5) et après (point 2) propagation de la substitution. Heureusement une substitution ne bouleverse pas la structure : une application substituée reste une application, un filtrage substitué reste un filtrage, etc. Évidemment, une variable substituée peut donner bien autre chose, mais ce cas est correctement géré :

$$\begin{aligned} [\![x\{x \leftarrow r\}]\!] &\overset{subst.}{=} [\![r]\!] \\ &\overset{subst.}{=} [\![x]\!]\{x \leftarrow [\![r]\!]\} \end{aligned}$$

Enfin, même si la même règle de définition de $[\![.]\!]$ est utilisée avant et après substitution, encore faut-il que cette règle produise des objets semblables. En particulier, la transformation $[\![I]\!]$ d'un inductif et celle $[\![\forall x : T, T']\!]$ d'un produit utilisent tous les deux un constructeur `mk_`s qui va dépendre d'une sorte s. Cette sorte est respectivement la sorte au bout de l'arité de I ou bien le type du produit. Il est alors indispensable pour la validité du présent lemme qu'une substitution ne puisse pas modifier cette sorte s. Ceci est trivial dans le cas de l'inductif : la sorte au bout de l'arité de I ne change pas par substitution. Par contre c'est beaucoup moins évident dans le cas du produit. Dans le système Cci d'origine, c'est même faux : si x : `Type`, alors $\forall y : Y$, x a pour type `Type`, alors que $(\forall y : Y, x)\{x \leftarrow$ `True`$\}$ a pour type `Prop`. Qu'en est-il dans le Cci$_m$? Si l'on détaille cet exemple, on obtient :

$[\![\forall y : Y, x]\!] =$ `mk_Type` $\forall y : [\![Y]\!]_{\mathbb{1}}$, $[\![x]\!]_{\mathbb{1}}$ ...

Pour être légale, la substitution doit maintenant être $\{x \leftarrow$ `True`$^{\dagger}\}$. On a alors :

$[\![\forall y : Y, x\{x \leftarrow$ `True`$^{\dagger}\}]\!]$
$= [\![\forall y : Y,$ `True`$^{\dagger}]\!]$
$=$ `mk_Type` $\forall y : [\![Y]\!]_{\mathbb{1}}$, (`Prop`$^{+}$`_Type`$^{+}$ $[\![$`True`$]\!]$).$\mathbb{1}$ ...

De l'autre côté, la substitution à appliquer après transformation est $\{x \leftarrow [\![$`True`$^{\dagger}]\!]\} = \{x \leftarrow$ `Prop`$^{+}$`_Type`$^{+}$ $[\![$`True`$]\!]\}$. On retrouve bien alors le même terme d'arrivée.

De façon générale, notre système modifié Cci$_m$ vérifie bien la conservation du type d'un produit par substitution. En effet, si s est le type d'origine, alors $s\{x \leftarrow r\} = s$ est bien un type du produit substitué d'après le lemme 4. Or il y a unicité des types dans Cci$_m$. □

2.4.6 Propriétés de réduction liées à la transformation $[\![.]\!]$

Comme nous avons exclu de notre étude les constantes et les « let-in », nous ne considérons ici que les réductions β et ι.

Théorème 7 *La transformation* $[\![.]\!]$ *préserve la réduction :* $t \to_{\beta\iota} t' \Rightarrow [\![t]\!] \to_{\beta\iota} [\![t']\!]$

PREUVE. Commençons tout d'abord par les cas où la réduction s'effectue en tête :

- $t = (\lambda u : U,\, a)\ b$ se réduisant par β en $t' = a\{u \leftarrow b\}$. On a alors $[\![t]\!]$ qui vaut $(\lambda u : [\![U]\!]_{\mathbb{1}},\, [\![a]\!])\ [\![b]\!]$ et peut bien se réduire vers $[\![a]\!]\{u \leftarrow [\![b]\!]\}$. Et ce dernier est bien égal via le lemme de substitution précédent à $[\![a\{u \leftarrow b\}]\!] = [\![t']\!]$.
- S'il s'agit d'une β-réduction en tête d'un schéma de types, on procède de même.
- $t = \mathtt{case}(C_i\ \overrightarrow{p}\ \overrightarrow{u},\, P,\, f_j)$ se réduisant par ι en $t' = (f_i\ \overrightarrow{u})$. On a alors $[\![t]\!] = \mathtt{case}(\overline{C}_i\ \overrightarrow{[\![p]\!]}\ \overrightarrow{[\![u]\!]},\, \overline{P},\, \overrightarrow{[\![f_j]\!]})$ qui peut bien se réduire vers $([\![f_i]\!]\ \overrightarrow{[\![u]\!]}) = [\![t']\!]$.
- $t = (\mathtt{fix}\ x : T := t_0)\ \overrightarrow{u}$ se réduisant par ι en $t' = t_0\{x \leftarrow (\mathtt{fix}\ x : T := t_0)\}\ \overrightarrow{u}$. On a alors $[\![t]\!] = (\mathtt{fix}\ x : [\![T]\!]_{\mathbb{1}} := [\![t_0]\!])\ \overrightarrow{[\![u]\!]}$. En particulier l'argument inductif de garde, qui commençait par un constructeur inductif, conserve sa structure de tête. On peut donc réduire $[\![t]\!]$ en $[\![t_0]\!]\{x \leftarrow (\mathtt{fix}\ x : [\![T]\!]_{\mathbb{1}} := [\![t_0]\!])\}\ \overrightarrow{[\![u]\!]}$. Le lemme de substitution précédent montre que ce terme est bien égal à $[\![t']\!]$.
- S'il s'agit d'une ι-réduction en tête d'un schéma de types, on procède de même.

Si maintenant la réduction a lieu à l'intérieur des termes, on procède par récurrence sur la structure de l'objet t d'origine. Comme ces cas sont sans surprise, nous ne les détaillerons pas. □

2.4.7 Validité des termes produits par la transformation $[\![.]\!]$

Théorème 8 *La transformation* $[\![.]\!]$ *préserve le typage :* $\Gamma \vdash t : T \ \Rightarrow\ [\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]_{\mathbb{1}}$

PREUVE. Par récurrence sur la dérivation de typage d'origine, en même temps que sur la bonne formation des contextes transformés $[\![\Gamma]\!]$.

(WF) $\mathcal{WF}(\varnothing) \Rightarrow \mathcal{WF}([\![\varnothing]\!])$

(WF) $$\frac{\Gamma \vdash U : s \quad u \notin \Gamma}{\mathcal{WF}(\Gamma; (u : U))} \Rightarrow \frac{\dfrac{[\![\Gamma]\!] \vdash [\![U]\!] : [\![s]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash [\![U]\!]_{\mathbb{1}} : s} \quad u \notin [\![\Gamma]\!]}{\mathcal{WF}([\![\Gamma]\!]; (u : [\![U]\!]_{\mathbb{1}}))}$$

En effet $[\![s]\!]_{\mathbb{1}} = s^+$ et donc la première projection de $[\![U]\!]$ a bien le type s.

(Ax) $$\frac{\mathcal{WF}(\Gamma) \quad s \in \{\mathtt{Set}, \mathtt{Prop}, \mathtt{Type}_i\} \quad i < j}{\Gamma \vdash s : \mathtt{Type}_j} \Rightarrow \frac{\mathcal{WF}([\![\Gamma]\!]) \quad s \in \{\mathtt{Set}, \mathtt{Prop}, \mathtt{Type}_i\} \quad i < j}{[\![\Gamma]\!] \vdash [\![s]\!] : [\![\mathtt{Type}_j]\!]_{\mathbb{1}}}$$

En effet $[\![s]\!] = \mathtt{mk\_Type}\ s^+\ \lambda\_,\ \lambda\_,\ \mathtt{True}$ et $[\![\mathtt{Type}_j]\!]_{\mathbb{1}} = \mathtt{Type}^+$.

(Var) $$\frac{\mathcal{WF}(\Gamma) \quad (u : U) \in \Gamma}{\Gamma \vdash u : U} \Rightarrow \frac{\mathcal{WF}([\![\Gamma]\!]) \quad (u : [\![U]\!]_{\mathbb{1}}) \in [\![\Gamma]\!]}{[\![\Gamma]\!] \vdash [\![u]\!] : [\![U]\!]_{\mathbb{1}}}$$

Tout d'abord, les variables sont invariantes par $[\![.]\!]$, donc $[\![u]\!] = u$. Et d'autre part, les types empilés dans l'environnement $[\![\Gamma]\!]$ sont bien de la forme $[\![U]\!]_{\mathbb{1}}$ (cf. règle (WF)).

$$\text{(Prod)}\ \frac{\Gamma \vdash T : s_1 \quad \Gamma; (x : T) \vdash T' : s_2 \quad \mathcal{P}(s_1, s_2, s_3)}{\Gamma \vdash \forall x : T,\, T' : s_3}$$

$$\Rightarrow$$

$$\frac{\llbracket\Gamma\rrbracket \vdash \llbracket T\rrbracket : \llbracket s_1\rrbracket_{\mathbb{1}} \qquad \llbracket\Gamma\rrbracket; (x : \llbracket T\rrbracket_{\mathbb{1}}) \vdash \llbracket T'\rrbracket : \llbracket s_2\rrbracket_{\mathbb{1}} \qquad \mathcal{PROD}(s_1, s_2, s_3)}{\dots}$$

$$\llbracket\Gamma\rrbracket \vdash \texttt{mk\_}s_3\ \ \forall x : \overline{T},\, \overline{T'}\ \ \lambda t,\, \lambda p,\, \forall x : \overline{T},\, \forall x' : \Lambda,\, (\widehat{T}\ x\ x') \to (\widehat{T'}\ (t\ x)\ (p\ x'))\ :\ \llbracket s_3\rrbracket_{\mathbb{1}}$$

Avec $\overline{T} = \llbracket T\rrbracket_{\mathbb{1}}$ et $\widehat{T} = \llbracket T\rrbracket_{\mathbb{2}}$ et idem pour T'. La dérivation ici esquissée est celle pour $s_3 \neq \texttt{Prop}$. Au lieu d'en écrire tous les détails, nous allons juste en décrire les grandes lignes. $\llbracket s_1\rrbracket_{\mathbb{1}} = s_1^+$ et $\llbracket s_2\rrbracket_{\mathbb{1}} = s_2^+$. Donc $\overline{T}$ et $\overline{T'}$ ont pour types respectifs s_1 et s_2, et $\widehat{T}$ et $\widehat{T'}$ ont pour types respectifs $\overline{T} \to \Lambda \to \texttt{Prop}$ et $\overline{T'} \to \Lambda \to \texttt{Prop}$. Une utilisation de la règle de typage du produit nous permet alors d'affirmer que $\forall x : \overline{T},\, \overline{T'}$ admet s_3 pour type. Il est alors facile de voir que la partie prédicat de la paire dépendante est bien de type $\forall x : \overline{T},\, \overline{T'} \to \Lambda \to \texttt{Prop}$. Il est alors bien légal de former cette paire dépendante, qui a donc effectivement pour type $\llbracket s_3\rrbracket_{\mathbb{1}} = s_3^+$.

Le cas $s_3 = \texttt{Prop}$ est une version simplifiée de ce qui précède, vu que $\texttt{mk\_}s_3$ n'a alors qu'un seul argument.

$$\text{(Lam)}\ \frac{\Gamma \vdash \forall u : U,\, V : s \quad \Gamma; (u : U) \vdash v : V}{\Gamma \vdash \lambda u : U,\, v : \forall u : U,\, V}$$

$$\Rightarrow$$

$$\frac{\dfrac{\llbracket\Gamma\rrbracket \vdash \llbracket\forall u : U,\, V\rrbracket : \llbracket s\rrbracket_{\mathbb{1}}}{\llbracket\Gamma\rrbracket \vdash \forall u : \llbracket U\rrbracket_{\mathbb{1}},\, \llbracket V\rrbracket_{\mathbb{1}} : s} \quad \llbracket\Gamma\rrbracket; (u : \llbracket U\rrbracket_{\mathbb{1}}) \vdash \llbracket v\rrbracket : \llbracket V\rrbracket_{\mathbb{1}}}{\llbracket\Gamma\rrbracket \vdash \lambda u : \llbracket U\rrbracket_{\mathbb{1}},\, \llbracket v\rrbracket : \llbracket\forall u : U,\, V\rrbracket_{\mathbb{1}}}$$

De l'hypothèse de récurrence concernant $\llbracket\forall u : U,\, V\rrbracket$, on déduit en prenant la première projection que $\llbracket\forall u : U,\, V\rrbracket_{\mathbb{1}} = \forall u : \llbracket U\rrbracket_{\mathbb{1}},\, \llbracket V\rrbracket_{\mathbb{1}}$ est de type s. Ceci plus l'autre hypothèse de récurrence concernant $\llbracket v\rrbracket$ permet d'appliquer la règle (Lam) et de conclure.

$$\text{(App)}\ \frac{\Gamma \vdash v : \forall u : U,\, V \quad \Gamma \vdash w : U}{\Gamma \vdash (v\ w) : V\{u \leftarrow w\}} \Rightarrow \frac{\llbracket\Gamma\rrbracket \vdash \llbracket v\rrbracket : \llbracket\forall u : U,\, V\rrbracket_{\mathbb{1}} \quad \llbracket\Gamma\rrbracket \vdash \llbracket w\rrbracket : \llbracket U\rrbracket_{\mathbb{1}}}{\llbracket\Gamma\rrbracket \vdash (\llbracket v\rrbracket\ \llbracket w\rrbracket) : \llbracket V\rrbracket_{\mathbb{1}}\{u \leftarrow \llbracket w\rrbracket\}}$$

Comme précédemment, on utilise l'égalité $\llbracket\forall u : U,\, V\rrbracket_{\mathbb{1}} = \forall u : \llbracket U\rrbracket_{\mathbb{1}},\, \llbracket V\rrbracket_{\mathbb{1}}$. Enfin, on a bien $\llbracket V\rrbracket_{\mathbb{1}}\{u \leftarrow \llbracket w\rrbracket\} = \llbracket V\{u \leftarrow w\}\rrbracket_{\mathbb{1}}$ via le lemme de substitution précédent.

$$\text{(Conv)}\ \frac{\Gamma \vdash U : s \quad \Gamma \vdash t : T \quad T =_{\beta\iota} U}{\Gamma \vdash t : U}$$

$$\Rightarrow$$

$$\frac{\dfrac{\llbracket\Gamma\rrbracket \vdash \llbracket U\rrbracket : \llbracket s\rrbracket_{\mathbb{1}}}{\llbracket\Gamma\rrbracket \vdash \llbracket U\rrbracket_{\mathbb{1}} : s} \quad \llbracket\Gamma\rrbracket \vdash \llbracket t\rrbracket : \llbracket T\rrbracket_{\mathbb{1}} \quad \llbracket T\rrbracket_{\mathbb{1}} =_{\beta\iota} \llbracket U\rrbracket_{\mathbb{1}}}{\llbracket\Gamma\rrbracket \vdash \llbracket t\rrbracket : \llbracket U\rrbracket_{\mathbb{1}}}$$

Et $T =_{\beta\iota} U$ implique bien $\llbracket T\rrbracket_{\mathbb{1}} =_{\beta\iota} \llbracket U\rrbracket_{\mathbb{1}}$, grâce au théorème précédent de préservation de la réduction par $\llbracket . \rrbracket$.

(CumT) $\dfrac{\Gamma \vdash t : \mathtt{Type}_i \quad i < j}{\Gamma \vdash t : \mathtt{Type}_j}$
$\Rightarrow$
$$\frac{[\![\Gamma]\!] \vdash [\![t]\!] : [\![\mathtt{Type}_i]\!]_{\mathbb{1}} \quad i < j}{[\![\Gamma]\!] \vdash [\![t]\!] : [\![\mathtt{Type}_j]\!]_{\mathbb{1}}}$$

Ceci est immédiat, car $[\![\mathtt{Type}_i]\!]_{\mathbb{1}} = [\![\mathtt{Type}_j]\!]_{\mathbb{1}} = \mathtt{Type}^+$.

(CumP) $\dfrac{\Gamma \vdash t : \mathtt{Prop}}{\Gamma \vdash t^\dagger : \mathtt{Type}}$
$\Rightarrow$
$$\frac{[\![\Gamma]\!] \vdash [\![t]\!] : [\![\mathtt{Prop}]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash [\![t^\dagger]\!] : [\![\mathtt{Type}]\!]_{\mathbb{1}}}$$

Ici $[\![t^\dagger]\!] = \mathtt{Prop}^+\mathtt{\_Type}^+\ [\![t]\!]$, qui est alors bien de type $[\![\mathtt{Type}]\!]_{\mathbb{1}} = \mathtt{Type}^+$

(CumS) $\dfrac{\Gamma \vdash t : \mathtt{Set}}{\Gamma \vdash t^\ddagger : \mathtt{Type}}$
$\Rightarrow$
$$\frac{[\![\Gamma]\!] \vdash [\![t]\!] : [\![\mathtt{Set}]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash [\![t^\ddagger]\!] : [\![\mathtt{Type}]\!]_{\mathbb{1}}}$$

Ici $[\![t^\ddagger]\!] = \mathtt{Set}^+\mathtt{\_Type}^+\ [\![t]\!]$, qui est alors bien de type $[\![\mathtt{Type}]\!]_{\mathbb{1}} = \mathtt{Type}^+$

(I-Type) Pour une arité $K = \forall \overrightarrow{u : U} s$, avec $s \neq \mathtt{Prop}$, on a :
$$\frac{\mathcal{WF}(\Gamma) \quad \mathtt{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (I : K) \in \Gamma_I}{\Gamma \vdash I : K}$$
$\Rightarrow$
$$\frac{\dfrac{\mathcal{WF}([\![\Gamma]\!]) \quad \mathtt{Ind}_n(\Gamma_{\overline{I},\widehat{I}} := \Gamma_{\overline{C},\widehat{C}}) \in [\![\Gamma]\!] \quad (\overline{I} : \overline{K}) \in \Gamma_I \quad (\widehat{I} : \widehat{K}) \in \Gamma_I}{\cdots}}{[\![\Gamma]\!] \vdash \lambda \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ (\mathtt{mk\_}s\ (\overline{I}\ \overrightarrow{u})\ (\widehat{I}\ \overrightarrow{u})) : [\![K]\!]_{\mathbb{1}}}$$

avec les abréviations $\overline{K} = \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s$ et $\widehat{K} = \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ (\overline{I}\ \overrightarrow{u}) \to \Lambda \to \mathtt{Prop}$. Et on a $[\![K]\!]_{\mathbb{1}} = \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s^+$. Les pointillés dans l'arbre de dérivation correspondent à deux utilisations de la règle (I-Type) pour déduire que $\overline{I}$ et $\widehat{I}$ ont pour types respectifs $\overline{K}$ et $\widehat{K}$, puis au typage des lambdas et de la paire dépendante.
Enfin, le cas $s = \mathtt{Prop}$ n'est qu'une simplification de ce qui précède, car $\mathtt{mk\_}s$ n'a alors qu'un seul argument.

(I-Cons) $\dfrac{\mathcal{WF}(\Gamma) \quad \mathtt{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (C : T) \in \Gamma_C}{\Gamma \vdash C : T}$
$\Rightarrow$
$$\frac{\mathcal{WF}([\![\Gamma]\!]) \quad \mathtt{Ind}_n(\Gamma_{\overline{I},\widehat{I}} := \Gamma_{\overline{C},\widehat{C}}) \in [\![\Gamma]\!] \quad (\overline{C} : [\![T]\!]_{\mathbb{1}}) \in \Gamma_C}{[\![\Gamma]\!] \vdash \overline{C} : [\![T]\!]_{\mathbb{1}}}$$

(I-WF) Concernant la déclaration d'un inductif I, d'arité $K = \forall \overrightarrow{u : U},\ s_I$ et de constructeurs $C_i : T_i$, si on pose $\Gamma_I = (I : K)$ et $\Gamma_C = (C_1 : T_1); \ldots; (C_k : T_k)$, on a alors à l'origine :

$$\frac{\Gamma \vdash K : s \quad \Gamma; \Gamma_I \vdash T_i : s_I \quad \mathcal{I}_n(\Gamma_I, \Gamma_C)}{\mathcal{WF}(\Gamma; \mathtt{Ind}_n(\Gamma_I := \Gamma_C))}$$

Examinons tout d'abord la définition de $\overline{I}$. Déjà, son arité est $\forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I$. Or on a :

$[\![\Gamma]\!] \vdash [\![\forall \overrightarrow{u : U},\ s_I]\!] : [\![s]\!]_{\mathbb{1}}$	hypothèse de récurrence associée à K
$[\![\Gamma]\!] \vdash \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I^+ : s$	par première projection, puisque $[\![s]\!]_{\mathbb{1}} = s^+$
$[\![\Gamma]\!] \vdash \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I : s$	car s_I^+ a un type plus grand que s_I.

Et pour les nouveaux types $[\![T_i]\!]_{\mathbb{1}}$ des constructeurs $\overline{C}_i$, on a :

$[\![\Gamma]\!]; [\![\Gamma_I]\!] \vdash [\![T_i]\!] : [\![s_I]\!]_{\mathbb{1}}$	hypothèse de récurrence associée à T_i
$[\![\Gamma]\!]; [\![\Gamma_I]\!] \vdash [\![T_i]\!]_{\mathbb{1}} : s_I$	par première projection

En fait, $[\![\Gamma_I]\!] = (I : \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I^+)$ et non pas $(\overline{I} : \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I)$. Mais I ne va apparaître dans les $[\![T_i]\!]_{\mathbb{1}}$ que derrière une première projection, sous une forme $(I\ \overrightarrow{w}).\mathbb{1}$. On peut alors changer ces occurrences en $(\overline{I}\ \overrightarrow{w})$ et remplacer $[\![\Gamma_I]\!]$ par $(\overline{I} : \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I)$. Enfin les conditions annexes $\mathcal{I}_n$ sont bien satisfaites pour $\overline{I}$. En particulier les $[\![T_i]\!]_{\mathbb{1}}$ sont bien toujours des types de constructeurs de sortes s_I. Quant à la condition de positivité, sans entrer dans les détails, nous donnerons juste comme intuition que $[\![.]\!]_{\mathbb{1}}$ préserve la structure des T_i, et donc a fortiori la positivité.

Passons maintenant à la définition de $\widehat{Q}$. Son arité est $\forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ (\overline{I}\ \overrightarrow{u}) \to \Lambda \to \mathtt{Prop}$, qui est bien typable de type `Type`. Quant aux constructeurs $\widehat{C}$, la définition choisie assure qu'ils sont bien des types de constructeurs pour $\widehat{I}$ de sorte `Prop`. Enfin, là encore nous ne détaillerons pas la vérification de la positivité, mais elle ne semble pas poser de problème.

(Case) Regardons maintenant le cas d'un filtrage sur un objet de type inductif I dont l'arité est $K = \forall \overrightarrow{p : P},\ K'$ avec $K' = \forall \overrightarrow{u : U},\ s$ et dont les constructeurs sont $C_i : T_i$. On va noter $T_i = \forall \overrightarrow{p : P},\ \forall \overrightarrow{v : V},\ (I\ \overrightarrow{p}\ \overrightarrow{w})$ les types des constructeurs des C_i. Soit enfin σ la substitution des paramètres formels $\overrightarrow{p}$ par des paramètres concrets q.

$$\frac{\Gamma \vdash e : I\ \overrightarrow{q}\ \overrightarrow{u} \quad \Gamma \vdash P : B \quad \mathcal{C}(I\ \overrightarrow{q} : K'_\sigma; B) \quad \forall i, \Gamma \vdash f_i : \forall \overrightarrow{v : V_\sigma},\ P\ \overrightarrow{w_\sigma}\ (C_i\ \overrightarrow{q}\ \overrightarrow{v})}{\Gamma \vdash \mathtt{case}(e,\ P,\ f_1 \ldots f_k) : P\ \overrightarrow{u}\ e}$$

$\Rightarrow$

$$\frac{[\![\Gamma]\!] \vdash [\![e]\!] : \overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![u]\!]} \qquad [\![\Gamma]\!] \vdash [\![P]\!] : [\![B]\!]_{\mathbb{1}} \qquad \mathcal{C}(\overline{I}\ \overrightarrow{[\![q]\!]} : \forall \overrightarrow{u : [\![U_\sigma]\!]_{\mathbb{1}}},\ s;\ \overline{B}) \quad \forall i, [\![\Gamma]\!] \vdash [\![f_i]\!] : \forall \overrightarrow{v : [\![V_\sigma]\!]_{\mathbb{1}}},\ [\![P\ \overrightarrow{w_\sigma}\ (C_i\ \overrightarrow{q}\ \overrightarrow{v})]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash \mathtt{case}([\![e]\!],\ \overline{P},\ [\![f_1]\!] \ldots [\![f_k]\!]) : [\![P\ \overrightarrow{u}\ e]\!]_{\mathbb{1}}}$$

En l'état, la version transformée n'est pas une application légale de la règle (Case) : il reste quelques ajustements à effectuer. Étudions tout d'abord le prédicat P, qui est de

la forme $\lambda\overrightarrow{u:U},\ \lambda x:(I\ \overrightarrow{q}\ \overrightarrow{u}),\ T$. Son type B est de la forme $\forall\overrightarrow{u:U},\ (I\ \overrightarrow{q}\ \overrightarrow{u})\to s_P$. Donc $[\![B]\!]_{\mathbb{1}} = \forall\overrightarrow{u:[\![U]\!]_{\mathbb{1}}},\ (\overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![u]\!]})\to s_P^+$. Et $[\![P]\!] = \lambda\overrightarrow{u:[\![U]\!]_{\mathbb{1}}},\ \lambda x:(\overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![u]\!]}),\ [\![T]\!]$. Ceci implique que notre nouveau prédicat $\overline{P} = \lambda\overrightarrow{u:[\![U]\!]_{\mathbb{1}}},\ \lambda x:(\overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{u}),\ [\![T]\!]_{\mathbb{1}}$ est bien typé, de type $\forall\overrightarrow{u:[\![U]\!]_{\mathbb{1}}},\ (\overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![u]\!]})\to s_P$. Notons $\overline{B}$ ce dernier type. Comme $\forall\overrightarrow{u:[\![U_\sigma]\!]_{\mathbb{1}}}$, s et $\overline{B}$ sont des arités de même sorte que leurs versions d'origine, la condition $\mathcal{C}$ est toujours vérifiée après transformation. Enfin on a les égalités suivantes :

$$[\![P\ \overrightarrow{u}\ e]\!]_{\mathbb{1}} = ([\![P]\!]\ \overrightarrow{[\![u]\!]}\ [\![e]\!]).\mathbb{1} = (\overline{P}\ \overrightarrow{[\![u]\!]}\ [\![e]\!])$$

et de même :

$$[\![P\ \overrightarrow{w_\sigma}\ (C_i\ \overrightarrow{q}\ \overrightarrow{v})]\!]_{\mathbb{1}} = (\overline{P}\ \overrightarrow{[\![w_\sigma]\!]}\ (\overline{C}_i\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![v]\!]}))$$

Ceci permet maintenant une application légale de la règle (Case), modulo quelques permutations entre la substitution $[\![.]\!]$ et σ.

(Fix) $$\frac{\Gamma\vdash T:s \quad \Gamma;(x:T)\vdash t:T \quad \mathcal{F}(x,T,k,t)}{\Gamma\vdash(\texttt{fix}\ x/k:T:=t)\ :\ T}$$

$\Rightarrow$

$$\frac{\dfrac{[\![\Gamma]\!]\vdash[\![T]\!]:[\![s]\!]_{\mathbb{1}}}{[\![\Gamma]\!]\vdash[\![T]\!]_{\mathbb{1}}:s} \quad [\![\Gamma]\!];(x:[\![T]\!]_{\mathbb{1}})\vdash[\![t]\!]:[\![T]\!]_{\mathbb{1}} \quad \mathcal{F}(x,[\![T]\!]_{\mathbb{1}},k,[\![t]\!])}{[\![\Gamma]\!]\vdash(\texttt{fix}\ x/k:[\![T]\!]_{\mathbb{1}}:=[\![t]\!])\ :\ [\![T]\!]_{\mathbb{1}}}$$

Concernant les conditions annexes $\mathcal{F}(x,[\![T]\!]_{\mathbb{1}},k,[\![t]\!])$:

- Le nombre des arguments attendus par $[\![t]\!]$ est le même que celui de t, et le k-ème argument de $[\![t]\!]$ a toujours un type inductif, qui est maintenant $\overline{I}$ au lieu de I.
- A chaque appel récursif $(x\ \overrightarrow{u})$ dans t va correspondre un (ou plusieurs) appel(s) récursif(s) $(x\ \overrightarrow{[\![u]\!]})$ dans $[\![t]\!]$. Le point important est que l'argument de « garde » u_k subit uniquement la transformation $[\![.]\!]$ pour devenir $[\![u_k]\!]$, ce qui ne change pas le fait qu'il soit structurellement plus petit que l'argument inductif d'origine, étant donné que $[\![.]\!]$ préserve la structure de t.

□

2.4.8 Correction de $\mathcal{E}$ vis-à-vis de la transformation $[\![.]\!]$

Commençons par deux résultats auxiliaires, qui confirment que les cas d'élimination de l'extraction sont bien valides vis-à-vis de $[\![.]\!]$.

Lemme 13 *Pour toute arité K de CCI_m bien typée dans un contexte Γ, on peut prouver $\forall x:[\![K]\!]_{\mathbb{1}},\ \forall x':\Lambda,\ [\![K]\!]_{\mathbb{2}}\ x\ x'$ dans le contexte $[\![\Gamma]\!]$.*

Preuve. Ceci se montre par récurrence sur la structure de K :

- Si $K = s$, alors $[\![K]\!]_{\mathbb{2}}$ est directement $\lambda x:[\![K]\!]_{\mathbb{1}},\ \lambda x':\Lambda,\ \texttt{True}$. Le terme $\lambda\_,\ \lambda\_,\ I$ convient alors comme preuve de la propriété recherchée.

- Si maintenant K est un produit $\forall x : T,\ K'$, on a alors :

  $[\![K]\!]_2 = \lambda t : [\![K]\!]_1,\ \lambda p : \Lambda,\ \forall x : [\![T]\!]_1,\ \forall x' : \Lambda,\ [\![T]\!]_2\ x\ x' \to [\![K']\!]_2\ (t\ x)\ (p\ x')$

  Or l'hypothèse de récurrence concernant K' nous donne un terme H_r dont le type est $\forall y : [\![K']\!]_1,\ \forall y' : \Lambda,\ [\![K']\!]_2\ y\ y'$ dans le contexte $[\![\Gamma]\!]; (x : [\![T]\!]_1)$. Le terme suivant convient alors :

  $\lambda z : [\![K]\!]_1,\ \lambda z' : \Lambda,\ \lambda x : [\![T]\!]_1,\ \lambda x' : \Lambda,\ \lambda\_,\ H_r\{x \leftarrow x\}\ (z\ x)\ (z'\ x')$

□

Lemme 14 *Pour tout type T de* CCI_m *admettant* `Prop` *comme type dans un contexte Γ, on peut prouver $\forall x : [\![T]\!]_1,\ \forall x' : \Lambda,\ [\![T]\!]_2\ x\ x'$ dans le contexte $[\![\Gamma]\!]$.*

Preuve. D'après le théorème 8, $[\![T]\!]$ a pour type $\texttt{Prop}^+$. Vu la définition de $\texttt{Prop}^+$, le prédicat $[\![T]\!]_2$ est donc $\lambda\_,\ \lambda\_,$ `True`. □

Nous allons maintenant énoncer et prouver le résultat principal de cette étude sémantique : l'extraction $\mathcal{E}$ entière est correcte vis-à-vis des prédicats de simulation obtenus via la transformation $[\![.]\!]$:

Théorème 9 *Pour tout objet t bien typé de* CCI_m*, si l'on a $\Gamma \vdash t : T$, alors $([\![T]\!]_2\ [\![t]\!]\ \mathcal{E}(t))$ est prouvable dans le contexte $[\![\Gamma]\!]_+$.*

Preuve. Tout d'abord, les cas de base que sont les arités et les parties logiques se traitent à l'aide des deux lemmes précédents. Le reste de la preuve se fait par récurrence sur la dérivation du typage $\Gamma \vdash t : T$.

(Ax) Dans cette règle permettant de typer les sortes, T est une sorte, et donc a fortiori une arité. On utilise donc le lemme 13.

(Prod) Ici encore T est une sorte, d'où l'utilisation du lemme 13.

(Var) $$\frac{\mathcal{WF}(\Gamma) \quad (x : T) \in \Gamma}{\Gamma \vdash x : T}$$

Si T admet `Prop` comme type ou est une arité, alors $\mathcal{E}(x) = \Box$ et on utilise le lemme 14 ou le lemme 13. Sinon $\mathcal{E}(x)$ est la variable de programme associée à x dans le contexte $[\![\Gamma]\!]_+$, à savoir x'. Et ce contexte contient alors une preuve H_x de type $([\![T]\!]_2\ x\ x')$, ce qui nous donne bien ici $([\![T]\!]_1\ [\![x]\!]\ \mathcal{E}(x))$.

(Lam) $$\frac{\Gamma \vdash \forall x : T_0,\ T' : s \quad \Gamma; (x : T_0) \vdash t' : T'}{\Gamma \vdash \lambda x : T_0,\ t' : \forall x : T_0,\ T'}$$

Si $T = \forall x : T_0,\ T'$ est une arité ou de type `Prop`, on applique un des lemmes précédents. Sinon, l'hypothèse de récurrence associée à t' affirme que :

$[\![\Gamma]\!]_+; (x : [\![T_0]\!]_1); (x' : \Lambda); (H_x : [\![T_0]\!]_2\ x\ x') \vdash H_R : [\![T']\!]_2\ [\![t']\!]\ \mathcal{E}(t')$

Or on a :

$[\![T]\!]_2 = \lambda z : [\![T]\!]_1,\ \lambda z' : \Lambda,\ \forall x : [\![T_0]\!]_1,\ \forall x' : \Lambda,\ \forall H_x : [\![T_0]\!]_2\ x\ x',\ [\![T']\!]_2\ (z\ x)\ (z'\ x')$

La propriété voulue s'écrit alors :

$\forall x : [\![T_0]\!]_{\mathbb{1}}, \forall x' : \Lambda, \forall H_x : [\![T_0]\!]_2\ x\ x', [\![T']\!]_2\ ((\lambda x : [\![T_0]\!]_{\mathbb{1}}, [\![t']\!])\ x)\ ((\lambda x' : \Box, \mathcal{E}(t'))\ x')$

Et ceci peut encore se simplifier via $(\lambda x : [\![T_0]\!]_{\mathbb{1}}, [\![t']\!])\ x = [\![t']\!]$ et $(\lambda x' : \Box, \mathcal{E}(t'))\ x' = \mathcal{E}(t')$. Finalement, le terme suivant convient comme preuve :

$\lambda x : [\![T_0]\!]_{\mathbb{1}}, \lambda x' : \Lambda, \lambda H_x : [\![T_0]\!]_2\ x\ x', H_r\{x, x', H_x \leftarrow x, x', H_x\}$

(App) $\dfrac{\Gamma \vdash t' : \forall x : T_0, T' \quad \Gamma \vdash t_0 : T_0}{\Gamma \vdash (t'\ t_0) : T'\{x \leftarrow t_0\}}$

Si $T'\{x \leftarrow t_0\}$ est une arité ou de type `Prop`, on applique un des lemmes précédents. Sinon, on va utiliser les deux hypothèses de récurrence :

$[\![\Gamma]\!]_+ \vdash H_r^1 : \forall x : [\![T_0]\!]_{\mathbb{1}}, \forall x' : \Lambda, [\![T_0]\!]_2\ x\ x' \to [\![T']\!]_2\ ([\![t']\!]\ x)\ (\mathcal{E}(t')\ x')$

$[\![\Gamma]\!]_+ \vdash H_r^2 : [\![T_0]\!]_2\ [\![t_0]\!]\ \mathcal{E}(t_0)$

Or la propriété voulue s'écrit :

$[\![T'\{x \leftarrow t_0\}]\!]_2\ [\![t'\ t_0]\!]\ \mathcal{E}(t'\ t_0) = [\![T']\!]_2\{x \leftarrow [\![t_0]\!]_{\mathbb{1}}\}\ ([\![t']\!]\ [\![t_0]\!])\ (\mathcal{E}(t')\ \mathcal{E}(t_0))$

Il suffit alors de prendre comme preuve :

$H_r^1\ [\![t_0]\!]\ \mathcal{E}(t_0)\ H_r^2$

(Conv) $\dfrac{\Gamma \vdash T : s \quad \Gamma \vdash t : T' \quad T' =_{\beta\iota} T}{\Gamma \vdash t : T}$

On a $[\![T']\!] =_{\beta\iota} [\![T]\!]$, et en particulier les deuxièmes projections sont convertibles. Il suffit donc de reprendre exactement le terme de preuve provenant de l'hypothèse de récurrence sur $t : T'$.

(CumT) $\dfrac{\Gamma \vdash t : \mathtt{Type}_i \quad i < j}{\Gamma \vdash t : \mathtt{Type}_j}$

$\mathtt{Type}_j$ est une sorte donc a fortiori une arité. On utilise le lemme 13.

(CumP) $\dfrac{\Gamma \vdash t : \mathtt{Prop}}{\Gamma \vdash t^\dagger : \mathtt{Type}}$

`Prop` est une sorte donc a fortiori une arité. On utilise le lemme 13.

(CumS) $\dfrac{\Gamma \vdash t : \mathtt{Set}}{\Gamma \vdash t^\ddagger : \mathtt{Type}}$

`Set` est une sorte donc a fortiori une arité. On utilise le lemme 13.

(I-Type) Un type inductif a nécessairement pour type une arité. On utilise donc le lemme 13.

(I-Cons) $\dfrac{\mathcal{WF}(\Gamma) \quad \mathtt{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (C : T) \in \Gamma_C}{\Gamma \vdash C : T}$

S'il s'agit d'un constructeur d'inductif logique, on utilise le lemme 14. Sinon, un dé-

pliage de $(\llbracket T \rrbracket_2\ \llbracket C \rrbracket\ \mathcal{E}(C))$ montre qu'il s'agit exactement du type du constructeur $\widehat{C}$ de l'inductif $\widehat{I}$. Il suffit alors de prendre ce $\widehat{C}$ comme terme de preuve.

(Case) Regardons maintenant le cas d'un filtrage sur un objet de type inductif I dont l'arité est $K = \forall \overrightarrow{p : P},\ K'$ avec $K' = \forall \overrightarrow{u : U},\ s$ et dont les constructeurs sont $C_i : T_i$. On va noter $T_i = \forall \overrightarrow{p : P},\ \forall \overrightarrow{v_i : V_i},\ (I\ \overrightarrow{p}\ \overrightarrow{w_i})$ les types des constructeurs des C_i. Soit enfin σ la substitution des paramètres formels $\overrightarrow{p}$ par des paramètres concrets q.

$$\frac{\Gamma \vdash e : I\ \overrightarrow{q}\ \overrightarrow{u} \quad \Gamma \vdash P : B \quad \mathcal{C}(I\ \overrightarrow{q} : K'_\sigma; B) \quad \forall i, \Gamma \vdash f_i : \forall \overrightarrow{v_i : V_{i\sigma}},\ P\ \overrightarrow{w_{i\sigma}}\ (C_i\ \overrightarrow{q}\ \overrightarrow{v_i})}{\Gamma \vdash \texttt{case}(e,\ P,\ f_1 \dots f_k) : P\ \overrightarrow{u}\ e}$$

Tout d'abord, si $T = (P\ \overrightarrow{u}\ e)$ admet `Prop` comme type, ou bien est une arité, on utilise l'un des lemmes précédents. Supposons maintenant qu'il s'agit d'un filtrage informatif. Les hypothèses de récurrence nous donnent :

$\llbracket \Gamma \rrbracket_+ \vdash H_r : \llbracket I \rrbracket_2\ \overrightarrow{\llbracket q \rrbracket}\ \overrightarrow{\llbracket u \rrbracket}\ \llbracket e \rrbracket\ \mathcal{E}(e)$ avec $\llbracket I \rrbracket_2 = \overline{I}$ si $s \neq \texttt{Prop}$ ou $\lambda \overrightarrow{\_}, \texttt{True}$ sinon

$\llbracket \Gamma \rrbracket_+ \vdash H_r^i : \llbracket \forall \overrightarrow{v_i : V_{i\sigma}},\ P\ \overrightarrow{w_{i\sigma}}\ (C_i\ \overrightarrow{q}\ \overrightarrow{v_i}) \rrbracket_2\ \llbracket f_i \rrbracket\ \mathcal{E}(f_i)$ pour tout i

Ou bien encore, avec les notations de la page 65 :

$\llbracket \Gamma \rrbracket_+ \vdash H_r^i : \forall \overrightarrow{\llbracket v_i : V_{i\sigma} \rrbracket},\ (\llbracket P \rrbracket\ \overrightarrow{\llbracket w_{i\sigma} \rrbracket}\ (\overline{C}_i\ \overrightarrow{\llbracket q \rrbracket}\ \overrightarrow{v_i})).2\ \ (\llbracket f_i \rrbracket\ \overrightarrow{v})\ \ (\mathcal{E}(f_i)\ \overrightarrow{v_i}^\Lambda)$

D'autre part on souhaite prouver :

$(\llbracket P \rrbracket\ \overrightarrow{\llbracket u \rrbracket}\ \llbracket e \rrbracket).2\ \ \texttt{case}(\llbracket e \rrbracket, \overline{P}, \overrightarrow{\llbracket f_i \rrbracket})\ \ \texttt{case}(\mathcal{E}(e), \Box, \overrightarrow{\mathcal{E}(f_i)})$

La tête e du filtrage est alors dans l'un des trois cas suivants :

- Son type inductif I est un inductif logique vide. On est alors dans une situation similaire à celle d'une preuve de `False` : il suffit d'éliminer cette preuve e via un filtrage pour pouvoir prouver n'importe quoi. Et la version $\llbracket e \rrbracket$ après transformation a toujours un type inductif vide, qui est maintenant $\overline{I}$. La preuve sera donc de la forme $\texttt{case}(\llbracket e \rrbracket, \dots,)$ avec le bon prédicat de tête.

- Son type inductif I est un inductif logique singleton. Il y a alors un seul constructeur C_1 à I, dont tous les arguments non-paramétriques sont logiques. Soit n le nombre de tels arguments. $\texttt{case}(\mathcal{E}(e), \Box, \overrightarrow{\mathcal{E}(f_1)})$ est en fait $\texttt{case}_n(\Box, \Box, \mathcal{E}(f_1))$, qui peut se réduire vers $(\mathcal{E}(f_1)\ \overrightarrow{\Box})$.

  Nous allons alors définir un nouveau prédicat $\widehat{P}$ valant :

  $\lambda \overrightarrow{u : \llbracket U \rrbracket},\ \lambda x : \overline{I}\ \overrightarrow{\llbracket q \rrbracket}\ \overrightarrow{u},\ (\llbracket P \rrbracket\ \overrightarrow{u}\ x).2\ \ \texttt{case}(x, \overline{P}, \overrightarrow{\llbracket f_i \rrbracket})\ \ (\mathcal{E}(f_i)\ \Box\ \dots\ \Box)$

  La preuve à construire va alors commencer par un filtrage sur $\llbracket e \rrbracket$ selon ce prédicat $\widehat{P}$. Ce filtrage n'a alors qu'une seule branche, dont le type doit être :

  $\forall \overrightarrow{v_1 : \llbracket V_1 \rrbracket_{\llbracket \sigma \rrbracket}},\ \widehat{P}\ \overrightarrow{\llbracket w_1 \rrbracket_{\llbracket \sigma \rrbracket}}\ (\overline{C}_1\ \overrightarrow{\llbracket q \rrbracket}\ \overrightarrow{v_1}) =$

  $\forall \overrightarrow{v_1 : \llbracket V_{1\sigma} \rrbracket},\ (\llbracket P \rrbracket\ \overrightarrow{\llbracket w_{1\sigma} \rrbracket}\ (\overline{C}_1\ \overrightarrow{\llbracket q \rrbracket}\ \overrightarrow{v_1})).2\ \ (\llbracket f_1 \rrbracket\ \overrightarrow{v_1})\ \ (\mathcal{E}(f_1)\ \overrightarrow{\Box})$

  via une ι-réduction et quelques permutations entre $\llbracket . \rrbracket$ et σ. Maintenant, pour chaque variable v_{1j} de la séquence $\overrightarrow{v_1}$, on sait qu'il s'agit d'une variable logique. En utilisant de manière répétée le lemme 14, on construit pour chaque j un terme

H_j de type $([\![V_{1j\sigma}]\!]_2\ v_{1j}\ \Box)$. La branche recherchée du filtrage est alors :

$\lambda\overrightarrow{v_1 : [\![V_{1\sigma}]\!]}, (H_r^1\ v_{11}\ \Box\ H_1\ \ldots\ v_{1n}\ \Box\ H_n)$

- Son type inductif I est informatif. Là encore, on va procéder par filtrage sur $[\![e]\!]$, mais il va falloir déconstruire H_r lors du filtrage. On commence donc par une généralisation (`Generalize` en `Coq`) vis-à-vis du type de H_r. Au niveau du terme de preuve final, ceci correspond à une application à H_r. Et la tête de cette application va maintenant être un filtrage sur $[\![e]\!]$ selon le prédicat $\widehat{P}$ suivant :

  $\lambda\overrightarrow{u : [\![U]\!]}, \lambda x : \overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{u},\ \widehat{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{u}\ x\ \mathcal{E}(e) \rightarrow$

  $([\![P]\!]\ \overrightarrow{u}\ x).2\quad \mathtt{case}(x, \overline{P}, \overrightarrow{[\![f_i]\!]})\quad \mathtt{case}(\mathcal{E}(e), \Box, \overrightarrow{\mathcal{E}(f_i)})$

  La j-ème branche de ce filtrage doit alors avoir pour type :

  $\forall\overrightarrow{v_j : [\![V_j]\!]_{[\![\sigma]\!]}}, \widehat{P}\ \overrightarrow{[\![w_j]\!]_{[\![\sigma]\!]}}\ (\overline{C}_j\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_j}) =$

  $\forall\overrightarrow{v_j : [\![V_{j_\sigma}]\!]}, \widehat{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![w_{j_\sigma}]\!]}\ (\overline{C}_j\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_j})\ \mathcal{E}(e) \rightarrow$

  $([\![P]\!]\ \overrightarrow{[\![w_{j_\sigma}]\!]}\ (\overline{C}_j\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_j})).2\ \ ([\![f_i]\!]\ \overrightarrow{v_i})\ \ (\mathtt{case}(\mathcal{E}(e), \Box, \overrightarrow{\mathcal{E}(f_j)}))$

  Cette branche du filtrage se construit alors ainsi :

  1. On commence par introduire les variables $\overrightarrow{v_j}$.
  2. On introduit la version spécialisée de H_r, où $[\![e]\!]$ est devenu $(\overline{C}_j\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_j})$.
  3. Il est alors possible d'inverser ce nouveau H_r (`Inversion` en `Coq`). Et compte tenu de la forme de son type, on obtient alors l'existence de variables v'_{jk} associées à chaque variable v_{jk} de $\overrightarrow{v_j}$, ainsi que des preuves H_k de type $([\![V_{jk\sigma}]\!]_2\ v_{jk}\ v'_{jk})$, et enfin le fait que $\mathcal{E}(e)$ est égal à $(C^j_\Lambda\ \overrightarrow{v_j}^\Lambda)$.
  4. On peut alors réécrire $\mathcal{E}(e)$ dans le but courant, dont la partie concernant $\mathcal{E}$ devient alors convertible à $(\mathcal{E}(f_j)\ \overrightarrow{v_j}^\Lambda)$.
  5. Il ne reste plus alors qu'à appliquer H_r^j aux bons arguments v_{jk}, v'_{jk} et H_k.

(Fix) $$\frac{\Gamma \vdash T : s \quad \Gamma; (f : T) \vdash t : T \quad \mathcal{F}(f, T, k, t)}{\Gamma \vdash (\mathtt{fix}\ f/k : T := t)\ :\ T}$$

Si T est de type `Prop` ou bien une arité, on utilise l'un des lemmes précédents. Sinon, on va utiliser l'hypothèse de récurrence :

$[\![\Gamma]\!]_+; (f : [\![T]\!]_\mathbb{1}; (f' : \Lambda); (H_f : [\![T]\!]_2\ f\ f') \vdash H_r : [\![T]\!]_2\ [\![t]\!]\ \mathcal{E}(t)$

Or on souhaite établir le but suivant :

$[\![T]\!]_2\ \ (\mathtt{fix}\ f : [\![T]\!]_\mathbb{1} := [\![t]\!])\ \ (\mathtt{fix}\ f' : \Box := \mathcal{E}(t))$

Par la suite, nous allons noter $\overline{\mathtt{fix}}$ et $\mathtt{fix}_\Lambda$ les points-fixes respectifs $(\mathtt{fix}\ f : [\![T]\!]_\mathbb{1} := [\![t]\!])$ et $(\mathtt{fix}\ f' : \Box := \mathcal{E}(t))$. L'idée est alors de faire une preuve par point-fixe, du style :

$\mathtt{fix}\ F : [\![T]\!]_2\ \overline{\mathtt{fix}}\ \mathtt{fix}_\Lambda := H_r\{f \leftarrow \overline{\mathtt{fix}}\}\{f' \leftarrow \mathtt{fix}_\Lambda\}\{H_f \leftarrow F\}$

Malheureusement, ce terme est légèrement inexact. En effet le type de H_r après substitution est $(\llbracket T \rrbracket_2\ \llbracket t \rrbracket \{f \leftarrow \overline{\texttt{fix}}\}\ \mathcal{E}(t)\{f' \leftarrow \texttt{fix}_\Lambda\})$ au lieu de $(\llbracket T \rrbracket_2\ \overline{\texttt{fix}}\ \texttt{fix}_\Lambda)$. Or les versions dépliées des points-fixes $\overline{\texttt{fix}}$ et $\texttt{fix}_\Lambda$ ne sont convertibles à leurs versions d'origine qu'en présence d'un argument de garde commençant par un constructeur. Il est en fait possible de corriger ces inexactitudes. Pour simplifier l'écriture, nous n'allons le faire que dans le cas où l'argument inductif de « garde » est en première position, mais cette méthode s'étend au cas général. On suppose donc que T s'écrit $\forall x : I\ \overrightarrow{u},\ T'$. Le type $(\llbracket T \rrbracket_2\ \llbracket t \rrbracket\ \mathcal{E}(t))$ de H_r se réécrit alors en :

$$\forall x : \overline{I}\ \overrightarrow{\llbracket u \rrbracket},\ \forall x' : \Lambda,\ \forall H_x : \widehat{I}\ \overrightarrow{\llbracket u \rrbracket}\ x\ x',\ \ \llbracket T' \rrbracket_2\ (\llbracket t \rrbracket\ x)\ (\mathcal{E}(t)\ x')$$

Or on peut prouver que :

$$\forall x : \overline{I}\ \overrightarrow{\llbracket u \rrbracket},\ \ \llbracket t \rrbracket \{f \leftarrow \overline{\texttt{fix}}\}\ x\ =\ \overline{\texttt{fix}}\ x$$

Il suffit pour établir cela de raisonner par cas sur x, et dans chacun des cas x est alors remplacé par un constructeur, ce qui fait que l'égalité devient triviale grâce à une ι-conversion. Et de même on peut prouver que :

$$\forall x : \overline{I}\ \overrightarrow{\llbracket u \rrbracket},\ \forall x' : \Lambda,\ \forall H_x : \widehat{I}\ \overrightarrow{\llbracket u \rrbracket}\ x\ x',\ \ \mathcal{E}(t)\{f' \leftarrow \texttt{fix}_\Lambda\}\ x'\ =\ \texttt{fix}_\Lambda\ x'$$

Là encore on procède par filtrage sur x. Ensuite pour en déduire la forme de x' on « inverse » H_x comme dans le cas du (case) précédent. Ceci nous montre alors que x' commence par un constructeur, et là encore l'égalité recherchée devient triviale après une ι-conversion. On peut donc utiliser ces deux égalités pour réécrire le type de la version substituée de H_r, afin que ce type devienne bien $(\llbracket T \rrbracket_2\ \overline{\texttt{fix}}\ \texttt{fix}_\Lambda)$.

Enfin, il faut vérifier que ce point-fixe de preuve que l'on vient de bâtir est bien légal : ses appels sont-ils bien structurellement décroissants ? Il s'agit alors de repérer les usages de H_f dans H_r. Or les occurrences de f dans le terme t d'origine sont appliquées à des arguments décroissants. Et la construction de H_r respecte la structure de t : un `case` produit un `case`, une application engendre une application, etc. A chaque application $(f\ y)$ dans t va correspondre une application $(H_f\ \llbracket y \rrbracket\ \mathcal{E}(y)\ \ldots)$ dans H_r, qui relie $(f\ \llbracket y \rrbracket)$ et $(f'\ \mathcal{E}(y))$. Et si y était structurellement plus petit que l'argument inductif d'origine, il est en de même pour $\llbracket y \rrbracket$.

□

2.5 Bilan des résultats de correction

Au terme de cette double étude de correction, il est temps de faire un bilan des propriétés de correction que nous avons établies. En fait, la partie syntaxique (section 2.3) et la partie sémantique (section 2.4) sont complémentaires.

- L'étude syntaxique permet d'établir la terminaison sans anomalie de toute réduction faible d'un terme extrait clos. Par contre si ce terme extrait est une fonction, ceci est peu révélateur. Il s'agit d'une méthodologie relativement simple, obtenue tôt au cours de cette thèse.

- Telle que nous l'avons formulée, notre analyse sémantique ne permet pas de retrouver la terminaison de nos programmes extraits. Peut-être cela serait-il possible en ajoutant des conditions de terminaison parmi les définitions de $[\![]\!]$, mais nous n'avons pas poussé plus loin dans cette direction faute de temps. En effet, cette analyse sémantique, qui a longtemps buté sur des questions comme la cumulativité de Cci, n'a finalement abouti que dans les derniers mois de cette thèse En tout cas, cette analyse permet de donner un sens aux fonctions extraites, et d'affirmer qu'elles préservent bien les propriétés des fonctions d'origine en Coq. Cette analyse ouvre également la porte à une étude de l'extraction en présence d'axiomes, chose exclue par notre première étude.

2.6 Vers une extraction plus réaliste

Nous avons vu auparavant que la réduction r_w étudiée dans la section 2.3.2 correspondait aux stratégies respectives de réductions de Haskell ou Ocaml, selon que l'on réduisait d'abord la tête ou l'argument des applications.

Il reste néanmoins quatre différences sensibles entre notre modèle théorique et les réductions effectivement utilisées par Haskell et Ocaml :

1. les inductifs vides et leurs éliminations
2. la règle $\rightarrow_i$ ad hoc d'élimination des singletons logiques
3. la règle $\rightarrow_\square$ ad hoc éliminant les arguments des constantes $\square$
4. la règle $\rightarrow_i$ concernant les points-fixes et leur concept de « garde »

2.6.1 Les inductifs vides

Nos langages cibles n'autorisent pas la définition d'inductifs vides, et encore moins le filtrage sur de tels inductifs. Mais en fait, si l'on manipule un objet ayant un type inductif vide, c'est que l'on est dans une situation impossible. Un filtrage sur un tel objet est en particulier une portion de code qui ne s'exécutera jamais.

Plus précisément, supposons que l'évaluation d'un objet extrait rencontre un terme de la forme `case`$(e, P, \varnothing)$. Le théorème 2 nous permet d'obtenir un terme Cci bien typé `case`$(e_0, P_0, \varnothing)$ correspondant. En particulier e_0 est un objet ayant pour type un inductif vide. On utilise alors un raisonnement similaire à celui utilisé lors de la preuve de ce théorème 2 : comme ces réductions se font de manière faible, *i.e.* dans un contexte vide et hors de tout lambda, l'objet inductif e_0 est donc clos. Il peut alors se réduire vers un terme ayant un constructeur en tête. Ceci est évidemment absurde, car un inductif vide ne possède par définition aucun constructeur.

On peut donc remplacer librement remplacer tout filtrage sur un objet inductif vide par du code arbitraire. En pratique, un choix naturel est de remplacer ce filtrage par la levée d'une exception. Ceci permet de souligner le caractère inaccessible de cet emplacement du code, tout en donnant au passage le type le plus général à cette portion de code. Nous avons ainsi utilisé en Ocaml la construction `assert false`, et `error` en Haskell.

Une fois n'est pas coutume, ce traitement d'un inductif vide est complètement indépendant du caractère logique ou informatif de l'inductif. Qu'il s'agisse de l'inductif logique `False` ou bien de son dual dans `Set` nommé `empty`, leurs deux éliminations `False_rec` et `empty_rec` se traduisent par la levée d'une exception.

2.6.2 L'élimination des singletons logiques

Dans l'étude de la section 2.3.2, nous avons utilisé une règle de ι-réduction adaptée pour traiter l'élimination de singletons logiques :

$$\texttt{case}_n(\Box,\, P,\, f) \rightarrow_\iota f\ \underbrace{\Box\ \ldots\ \Box}_{n}$$

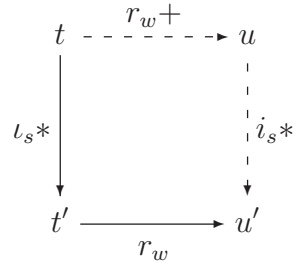
La raison d'être de cette règle ad hoc est de permettre à la fonction $\mathcal{E}$ de perturber le moins possible la structure des termes, ce qui simplifie son étude. Cette réduction n'a donc rien de fondamentale, au point qu'il est parfaitement possible de l'intégrer à l'extraction : plutôt que d'essayer d'encoder cette règle spéciale dans nos langages cibles, nous allons « pré-compiler » dès l'extraction toutes ces éliminations singletons logiques, sans perdre pour autant la correction.

Justifier une telle optimisation des termes extraits n'est pas totalement immédiat. En effet cela implique de réduire toutes les éliminations singletons logiques, y compris celles situées sous des lambdas, alors que nos résultats de corrections de la section 2.3.2 ne concernent que l'utilisation de la version *faible* de cette ι-réduction.

Il peut d'ailleurs sembler paradoxal de vouloir effectuer des réductions fortes après avoir mis en garde contre de telles réductions dans les exemples du début de la section 2.3.2. Mais considérons de nouveau la fonction `cast` de la page 51, qui se traduit en un filtrage sur un inductif singleton logique. Ce n'est pas tant la ι-réduction du corps de `(cast H 0)` vers `0` qui génère une erreur dans l'exécution de la fonction `exemple`, mais plutôt le filtrage suivant sur `0` vu comme un booléen. Et ce second filtrage restera bien interdit car il se fait sous un lambda.

Nous allons noter $\rightarrow_{\iota_s}$ la version *forte* de cette ι-réduction ad hoc, utilisable y compris sous les lambdas, à l'inverse de la version faible $\rightarrow_{\iota_w}$. À partir d'un terme extrait t, nous allons maintenant justifier la correction d'un terme t' obtenu via $t \rightarrow_{\iota_s}^* t'$, en procédant par simulation avec t.

Théorème 10 *Soient t, t' et u' trois termes de* $\text{CCI}_\Box$ *tels que $t \rightarrow_{\iota_s}^* t'$ et $t' \rightarrow_{r_w} u'$. Il existe alors un terme* $\text{CCI}_\Box$ *u tel que $u \rightarrow_{\iota_s}^* u'$ et $t \rightarrow_{r_w+} u$.*

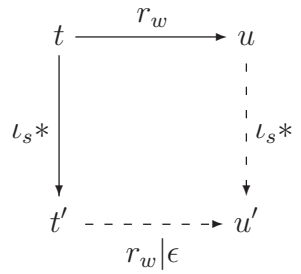


PREUVE. Compte tenu de la règle $\rightarrow_\iota$ pour les singletons logiques, les termes t et t' ont des structures extrêmement voisines. Il suffit alors de comparer la position du redex r que l'on réduit dans t' avec la position correspondante dans t.

- Si cette position est en dehors de toute branche de filtrage sur un singleton logique, alors la même réduction peut être effectuée dans t, ce qui nous donne un terme u convenable.
- Supposons maintenant que cette position dans t se trouve sous un ou plusieurs filtrages singletons logiques. Considérons tout d'abord le cas d'un unique filtrage singleton logique sur le chemin vers le redex correspondant à r dans t. La réduction que l'on effectue dans t' est faible. Dans t, cela signifie que notre filtrage singleton logique est a fortiori situé hors de tout lieur, qu'il s'agisse de lambdas ou de filtrages. On peut commencer par utiliser $\rightarrow_{\iota_w}$ pour réduire ce filtrage singleton logique, avant de réduire le redex correspondant à r. Et dans le cas de multiples filtrages singletons logiques, l'argument est le même : le plus externe de ces filtrages peut être réduit de façon faible, puis le second, et ainsi de suite, avant de terminer par le redex correspondant à r.

□

Théorème 11 *Soient t, t' et u trois termes de* CCI$_\square$ *tels que $t \rightarrow_{\iota_s}^* t'$ et $t \rightarrow_{r_w} u$. Il existe alors un terme* CCI$_\square$ *u' tel que $u \rightarrow_{\iota_s}^* u'$ et vérifiant $t' \rightarrow_{r_w} u'$ ou $t' = u'$.*



PREUVE. Tout d'abord, si la réduction de t vers u est une ι-réduction de singleton logique déjà effectuée entre t et t', alors prendre $u' = t'$ convient. Sinon la réduction faible entre t et u se fait hors de tout filtrage, y compris de filtrage singleton logique. Il y a donc un redex exactement identique dans t', que l'on peut réduire pour trouver un u' convenable. □

Outre les réductions $\rightarrow_{r_w}$, les réductions $\rightarrow_\square$ peuvent également être simulées entre le terme extrait t d'origine et sa version optimisée t'. Au final, toute suite de réductions partant de t' termine, sinon le théorème 10 nous donnerait une suite infini de réductions pour le terme extrait t. Et en combinant les deux théorèmes précédents on montre que les formes normales faibles de t et t' sont reliées par $\rightarrow_{\iota_s}$, et en particulier égales si elles ne contiennent plus de λ-abstractions.

Lors de l'extraction, on est donc bien en droit d'effectuer toutes les ι-réductions singletons logiques, même fortes, sur le terme extrait brut engendré par $\mathcal{E}$. De plus la tête d'un filtrage singleton logique a forcément `Prop` comme sorte, et donc est extrait par $\mathcal{E}$ vers $\square$. Ceci fait donc que tous les filtrages singletons logiques $\texttt{case}_n$ sont bien réduits par $\rightarrow_{\iota_s}$ et disparaissent complètement.

2.6.3 L'élimination des arguments éventuels de □

Considérons maintenant la réduction ad hoc $(\Box\ x) \to_\Box$. À la différence de la réduction ad hoc précédente sur les singletons logiques, il ne va pas être possible de s'en passer complètement après extraction. En effet, l'extraction $\mathcal{E}$ n'engendre jamais elle-même de tels termes $(\Box\ x)$, puisque le terme entier d'origine $(f\ x)$ est alors identifié comme étant à éliminer, et donne directement □. Par contre, l'exemple 1 de la page 49 montre que de tels termes peuvent apparaître à la suite de réductions sur un terme extrait.

Ocaml

Si l'on évalue comme Ocaml les termes extraits avec une stratégie stricte, Il faut alors s'arranger pour que cette application $(\Box\ x)$ n'échoue pas. En pratique cela se peut se faire en choisissant convenablement le terme concret qui va implanter □. Usuellement, on choisit le terme `()` de type `unit` pour remplacer une constante arbitraire comme notre □, mais cela ne convient donc pas ici. On peut alors être tenter d'utiliser plutôt `fun _ → ()`, ou encore `fun _ _ → ()` si l'on sait que □ peut recevoir deux arguments, et ainsi de suite. Malheureusement, il nous a été impossible de déterminer simplement le nombre maximal d'arguments que peut recevoir un terme □ donné. Nous avons donc opté pour une solution plus générale quoique moins élégante, à savoir un point-fixe absorbant ses arguments. En Ocaml, cela peut s'écrire ainsi :

```
let rec f x = f
```

Bien sûr, cette définition est mal typée, et nous verrons au prochain chapitre comment contourner ce problème. En tout cas, du point de vue de l'exécution, une constante □ implantée de cette façon peut recevoir un nombre arbitraire d'arguments sans échouer.

Haskell

Dans le cas d'une évaluation paresseuse à la Haskell, on pourrait parfaitement reprendre cette idée d'un point-fixe absorbant ses arguments. Mais cela n'est en fait pas nécessaire. Si l'on étudie de nouveau l'exemple 1, l'objet extrait puis réduit est $\lambda g : \Box, (g\ (\Box\ \mathtt{0}))$. Le sous-terme $(\Box\ \mathtt{0})$ ne s'y trouve pas en tête, et la fonction g peut tout à fait ne pas utiliser son argument, auquel cas l'application $(\Box\ \mathtt{0})$ ne sera jamais calculée. Cette situation n'a rien d'exceptionnelle, on n'aura en fait jamais besoin d'utiliser la règle $\to_\Box$ avec la stratégie paresseuse. En effet cette paresseuse revient à toujours réduire la tête des termes, d'après les deux règles suivantes de compatibilité (voir la définition 13) :

$$\frac{u \to_? v}{(u\ t) \to_? (v\ t)} \qquad \frac{u \to_? v}{\mathtt{case}(u, P, \ldots) \to_? \mathtt{case}(v, P, \ldots)}$$

Prenons un terme d'origine t_0 clos et bien typé dans le Cci, et $t = \mathcal{E}(t_0)$. Soit maintenant u un des réduits successifs de t par la stratégie paresseuse. Si une réduction $(\Box\ x) \to_\Box \Box$ peut intervenir dans u à une position permise par ces deux règles de compatibilité, il y a deux situations possibles :

- Le □ peut être au sommet de u, qui est alors de la forme $(\square\ \overrightarrow{a})$. D'après le théorème 2, ce terme correspond alors à un terme de $u_0 = (f_0\ \overrightarrow{a_0})$ de CCI$_\square$, qui est un réduit du terme t d'origine. Et f_0 est alors soit de sorte `Prop` ou soit un schéma de types. D'après le lemme de stabilité 1, il en est de même pour le terme entier $u_0 = (f_0\ \overrightarrow{a_0})$. Ensuite le lemme 2 montre que le terme d'origine t_0 est également de sorte `Prop` ou bien un schéma de types. Et donc $\mathcal{E}(t_0) = \square$, ce qui montre que le cas considéré ici est impossible.
- Le □ peut être au début de la tête d'un filtrage, c'est-à-dire que u contient un sous-terme de la forme `case`$(\square\ \overrightarrow{a}, \ldots, \ldots)$. Un raisonnement similaire à celui du cas précédent montre que le terme inductif correspondant dans CCI à $(\square\ \overrightarrow{a})$ est de sorte `Prop`. Mais dans le terme tout juste extrait $t = \mathcal{E}(t_0)$, tout `case` sur un inductif logique a forcément □ comme tête, qu'il s'agisse d'un inductif logique vide ou bien d'un singleton logique. Et cela ne peut changer par réduction, ce qui contredit la présence dans u du sous-terme de la forme `case`$(\square\ \overrightarrow{a}, \ldots, \ldots)$.

La règle $\rightarrow_\square$ n'est donc jamais utilisée dans une réduction à la Haskell. □ peut donc toujours être implanté par un terme arbitraire.

En fait on peut même aller plus loin et considérer que □ est un résultat final anormal lors de l'évaluation d'un terme extrait. Le calcul d'un terme extrait n'a en effet d'intérêt que s'il fournit un résultat ... calculatoire, comme par exemple `true`, `3`, ou encore `fun x⇒x`. Ce résultat peut tout à fait contenir des résidus □ de parties logiques ou de schémas de types, par exemple s'il s'agit d'une fonction. Mais que le résultat entier soit □ est anormal. Nous avons donc fait le choix d'implanter □ par la levée d'une exception. Et cela ne perturbe en rien les calculs sur des termes informatifs :

- Nous venons de voir que □ ne peut être évalué comme tête d'une application.
- □ ne sera jamais non plus évalué comme tête d'un filtrage sur un terme inductif logique, puisque nous avons vu dans les deux sections précédentes comment faire disparaître les filtrages sur les inductifs vides et ceux sur les inductifs singletons logiques.

2.6.4 Vers une réduction usuelle des points-fixes

La réduction des points-fixes est la dernière différence majeure entre les réductions de notre système CCI$_\square$ et celles des langages cibles. En effet il n'y a évidemment pas dans ces langages cibles de concept d'argument de « garde », devant commencer par un constructeur avant que la réduction soit possible.

Ocaml

Dans le cas d'Ocaml, la différence n'est pas si grande. En effet un point-fixe à deux arguments `let rec f x y = t` ne sera réduira que s'il reçoit ses deux arguments. Et dans le cas d'un argument inductif, on va évaluer complètement cet argument, dont la valeur commencera donc bien par un constructeur, avant de déplier le point-fixe. On peut alors tout simplement traduire l'exemple `fix` $f\ \{f/2 : \square := \lambda x, \lambda y, t\}$ par `let rec f x y = t in f`. Et il n'est pas abusif de supposer que le corps du point-fixe commence par suffisamment

de λ-abstractions, ici au moins deux. En effet la syntaxe concrète de Coq pour le `Fixpoint` et le `fix` nous oblige à déclarer les arguments au moins jusqu'à l'argument de garde. Enfin, traiter les points-fixes mutuels ne pose pas plus de difficultés. Il n'y a pas non plus de soucis avec le cas d'un argument de garde logique, la condition « être $\Box$ » remplaçant juste la condition « commencer par un constructeur ». L'évaluation complète d'un argument logique ne peut en effet terminer que sur $\Box$.

Haskell

Par contre, pour Haskell, la situation est plus délicate. Le dépliage d'un point-fixe `f` se fait lorsque `f` arrive en tête du terme à évaluer, et se fait sans réduction préalables des arguments. La contrainte sur l'évaluation au moins partielle de l'argument de garde avant tout dépliage n'a donc aucune raison d'être respectée. Il n'est donc pas sûr que l'ordre des réductions en Haskell puisse être reflété au niveau Cci dans l'esprit du théorème 2.

On peut tenter de se convaincre informellement que tout se passe correctement. Les contraintes de décroissance que le Cci impose font que l'appel récursif suivant se fait avec un argument de garde « plus petit ». Et pour obtenir un tel argument « plus petit », il est nécessaire de déconstruire au moins un niveau de l'argument d'origine. L'évaluation de l'argument de garde est donc simplement repoussé d'appel récursif en appel récursif.

L'argumentaire précédent est insatisfaisant. Déjà, une telle modification de l'ordre d'évaluation n'est pas nécessairement anodine. De plus notre raisonnement suppose que l'analyse de l'argument de garde, destinée à produire l'argument suivant, se fait en tête de terme, et qu'Haskell l'effectuera donc sans délai. Habituellement, il est vrai que le corps d'un point-fixe commence immédiatement par un `match`. Mais cela n'est pas obligatoire, comme le montre l'exemple de `Acc_iter` page 30. Dans ce cas, l'analyse de l'argument de garde est repoussée *dans* le terme servant d'argument suivant. `Acc_iter` est en fait un mauvais exemple, car son argument de garde est logique, et disparaît donc à l'extraction. Mais dans un exemple similaire avec un argument de garde informatif, cet argument pourrait très bien ne jamais être évalué par Haskell.

Nous allons maintenant esquisser une justification rigoureuse de la traduction simple des points-fixes $\text{Cci}_{\Box}$ vers les points-fixes Haskell. Nous allons pour cela montrer que l'ordre des calculs lors d'une réduction Haskell d'un point-fixe extrait peut être simulé dans le Cci à l'aide d'une version modifiée du point-fixe d'origine. Pour cela nous allons utiliser une remarque de C. Paulin faite p. 103 de [68] : tout point-fixe structurel sur un inductif I peut se transformer en une définition par récursion bien-fondée sur un ordre $<_I$, c'est-à-dire en fait en un point-fixe structurel sur l'inductif `Acc` ayant pour paramètre $<_I$. L'intérêt de cette traduction est de rajouter un argument d'accessibilité logique qui va servir de nouvel argument de garde. De la sorte, il n'y a plus besoin dans Cci d'évaluer même partiellement les arguments informatifs du point-fixe avant un dépliage, les contraintes étant concentrées sur le nouvel argument logique.

Nous n'allons pas proposer ici de démonstration de la remarque de C. Paulin, déjà justifiée dans [68]. À la place, nous allons tenter de l'illustrer sur un exemple simple, celui de l'addition sur les entiers de Peano. Pour $I =$ `nat`, un ordre $<_I$ qui va convenir à nos besoin est l'ordre standard `lt` (ou `<`) sur les entiers. Nous allons également utiliser la version large `le` (ou $\leq$)

de cet ordre. Voici donc une version de `plus` dans laquelle l'argument de garde n'est pas un des deux arguments entiers, mais un argument d'accessibilité logique :

```
Definition plus (n m:nat) : nat :=
  (fix plusrec (n m:nat) (a:Acc lt n) {struct a} : nat :=
      match n as n0 return n0≤n → nat with
       | 0 ⇒ fun h ⇒ m
       | (S n') ⇒ fun h ⇒ S (plusrec n' m (Acc_inv a n' h))
      end (le_refl n))
  n m (lt_wf n).
```

Comparons avec la dernière version de `plus` présentée page 20 :

- On a ajouté un argument supplémentaire d'accessibilité `a`, qui sert d'argument de décroissance. Et lors de l'appel récursif, le nouvel argument d'accessibilité est obtenu via la fonction `Acc_inv` déjà rencontrée page 30.
- Le véritable point-fixe, `plusrec`, attend maintenant trois arguments. La fonction `plus` est donc une encapsulation de ce `plusrec` dans laquelle on fournit la preuve `(lt_wf n)` de l'accessibilité de `n` comme troisième argument.
- Enfin, le typage du `match` est bien plus complexe que dans le `plusrec` d'origine, et nécessite des annotations particulières, ainsi qu'une abstraction artificielle sur une variable `h` de type `n≤n`. Et une preuve de ce `n≤n` est fournie immédiatement après ce `match` via le terme `(le_refl n)`.

Pour se convaincre que ce nouveau `plus` permet de simuler au niveau Cci toute réduction à la Haskell de l'extraction du `plus` d'origine, il suffit de comparer les extractions des deux `plus`. Voici en particulier ce que donne $\mathcal{E}$ sur notre nouveau `plus`[4] :

```
Definition plus (n m:□) : □ :=
  (fix plusrec (n m:□) (_:□) : □ :=
      match n with
       | 0 ⇒ fun _ ⇒ m
       | (S n') ⇒ fun _ ⇒ S (plusrec n' m □)
      end □)
  n m □.
```

On constate la présence de résidus de parties logiques, qui différencient cette extraction de celle du `plus` d'origine. Mais ces résidus n'ont pas d'influence sur l'ordre des calculs. Par exemple, les abstractions anonymes présentes dans les branches du `match` reçoivent immédiatement leur argument □ situé après le `match`. Tout calcul à la Haskell avec l'extraction d'origine correspond donc à un calcul avec cette nouvelle extraction, calcul qui peut enfin être simulé au niveau Cci, ce qui garantit la correction de l'extraction initiale.

Notons enfin que dans l'extraction du nouveau `plus`, tous ces résidus logiques, constantes □ et abstractions anonymes, sont en pratique détectés et supprimés par les optimisations

[4]Nous avons ici conservé une syntaxe à la Coq, plus lisible, au lieu de notre syntaxe Cci$_\square$.

de l'extraction que nous décrivons en section 4.3. Au final le terme extrait de ce `plus` est *exactement* le même que celui extrait de l'ancien `plus`.

Chapitre 3

Le typage des termes extraits

Nous avons construit dans le chapitre précédent une extraction produisant à partir de termes Coq des termes non typées. Nous avons en particulier montré que l'exécution de ces termes extraits était nécessairement finie et sans erreur. Il reste maintenant à étudier la traduction finale vers un véritable langage fonctionnel, ce qui est l'objet du présent chapitre.

Comme nous l'avons mentionné en introduction, nous souhaitons que le code extrait puisse être intégré à un développement plus large. Il est donc nécessaire de pouvoir au moins comprendre la signature des objets extraits. Deux choix se présentent alors : on peut soit générer du code-source pour un langage particulier, ou bien directement du byte-code ou du code assembleur associé avec une interface lisible. Mais outre la difficulté de générer du code binaire, ceci conduirait à une solution « boîte noire », sans possibilité de contrôle a posteriori. Nous avons préféré produire du code source, ce qui laisse à l'utilisateur la possibilité de lire ce code, et qui permet aussi de bénéficier alors des compilateurs optimisants déjà existants. Le mouvement « open source » a en particulier montré que la lisibilité des sources accroît grandement la confiance dans un programme.

De ce premier choix découle une nouvelle question : quel langage doit-on utiliser comme cible de l'extraction ? Tout ce qu'il nous faut, c'est un λ-calcul avec des types inductifs. Ceci explique le choix de langages dérivés de ML, à savoir Ocaml et Haskell. Mais ces langages sont typés, et leur systèmes de types à la Hindley-Milner [59, 27] sont sensiblement différents de celui de Coq. En particulier on ne peut exprimer dans ces langages des types dépendants ou des univers. Comme mentionné précédemment, l'ancienne extraction faisait déjà le choix pragmatique de refuser tout terme Coq impliquant la sorte Type. Mais même cette restriction n'était pas suffisante pour être certain d'obtenir des termes ML bien-typés. Par exemple, la notion de polymorphisme diffère entre ML et Coq. Il est donc clair qu'une traduction simpliste des λ-termes Coq vers des λ-termes ML peut aboutir à des termes non-typables.

Il va donc falloir adapter nos termes si l'on veut utiliser un compilateur standard comme celui de Ocaml ou Haskell, non modifié pour l'extraction. En même temps, nous souhaiterions rester aussi proche que possible d'une traduction directe, et ce pour plusieurs raisons. Déjà, une grande majorité des exemples ordinaires de termes Coq ont bien une contrepartie ML correctement typée. Ensuite, le besoin d'une interface au code extrait milite également en faveur d'une traduction simple et naturelle. Enfin, tout effort visant à contourner ces problèmes de typage par un encodage ad hoc semble aboutir à une étape de pré-compilation que l'on souhaite justement éviter. Un encodage particulier a été testé par L. Pottier [70],

mais ce codage continue à pouvoir produire des termes ML non-typables.

Comment alors utiliser des compilateurs pour langages typés avec des termes potentiellement non-typables ? En ce qui concerne Ocaml, nous utilisons maintenant, tout comme L. Pottier, une fonctionnalité non-documentée de ce langage, à savoir `Obj.magic`. Cette fonction attribue un type générique `'a` à n'importe quel terme. En utilisant cette fonction, on peut donc contourner localement la vérification des types par le compilateur Ocaml. Nous verrons dans un deuxième temps comment l'extraction actuelle permet de générer automatiquement des `Obj.magic` dans le code extrait. Pour ce qui est de Haskell, certaines des implantations de ce langage proposent une fonction non-documentée `unsafeCoerce` qui semble équivalente à `Obj.magic`. Nous pensons donc pouvoir étendre par la suite à Haskell cette génération automatique de termes artificiellement bien-typés.

Mentionnons au passage la réalisation d'une extraction expérimentale vers le langage Scheme. Comme ce langage fonctionnel issu de Lisp est non-typé, il nous a semblé un moment être un langage cible prometteur pour l'extraction, étant donné que le problème des termes non-typables ne se posait pas ici. Malheureusement, Scheme ne possède pas nativement de types inductifs et de filtrage sur de tels types. Il est donc nécessaire de les encoder à l'aide de macros, sauf dans le cas particulier de l'implantation Bigloo [75]. En outre, des tests préliminaires ont montré une différence importante d'efficacité du code extrait en faveur de Haskell/Ocaml au détriment de Scheme. L'effort en faveur de cette extraction Scheme n'a donc pas été poursuivi.

D'autre part, il n'est même pas suffisant d'assurer simplement l'existence d'un type correct pour chaque terme extrait si l'on ne peut préciser duquel il s'agit à l'avance. En effet, comme l'on veut permettre une intégration simple du code extrait dans des développements plus larges, nous devons être en mesure de prévoir quels vont être les types des termes extraits, et de produire des fichiers interfaces. De plus, si l'on veut pouvoir extraire les modules et foncteurs de Coq (cf. section 4.1), il va falloir être capable de se livrer à de tels calculs de types.

Cette étude du typage des termes extraits est organisée comme suit. Nous allons tout d'abord décrire les erreurs de typages que notre extraction sur les termes peut produire. Puis nous présenterons une méthode automatique de contournement de ces erreurs, via l'insertion dans le code de primitives de changement de type, permettant de le rendre artificiellement typable. Pour guider cette insertion, on calcule tout d'abord un type extrait « raisonnable » à partir du type Coq initial de l'objet considéré. Nous décrirons cette extraction $\widehat{\mathcal{E}}$ sur les types. Ensuite, on force ce type extrait souhaité à devenir réellement celui du terme extrait brut, initialement non-typé :

$$t : T \Rightarrow \mathcal{E}(t) : \widehat{\mathcal{E}}(T)$$

Ceci se fait grâce à une variante de l'algorithme $\mathcal{M}$ de vérification/inférence de types, modifié pour que chaque erreur de typage détectée produise une insertion d'un changement de type.

3.1 Analyse des problèmes de typage

Les problèmes commencent avec les déclarations de types, qu'il s'agisse des types inductifs ou des abréviations. En effet Coq comporte des fonctionnalités de typage sans contreparties

en Ocaml ou Haskell, dont en particulier les suivantes :

- filtrages au niveau des types (`match`)
- points-fixes au niveau des types (`fix`)
- polymorphisme par quantifications universelles à des positions non prénexes ou dans les types des constructeurs inductifs.

Nous allons détailler dans la suite chacune de ces situations. Dans l'ancienne extraction des versions 6.x et antérieurs, ces déclarations de types non-traduisibles étaient refusées avec le message du genre : `Error: .... is not an ML type`.

La situation pour les termes est différente. En effet l'utilisation d'un type non-traduisible dans un terme Coq n'empêche pas, a priori, sa typabilité en Haskell ou Ocaml, puisque ces langages ne comportent pas d'annotations de types sur les lambdas et sur les filtrages : il n'y a donc pas de référence explicite aux types intraduisibles. L'ancienne extraction générait alors du code sans se préoccuper de sa typabilité. L'inférence de type sur ce terme extrait pouvait alors soit trouver un type plus simple, acceptable en Haskell ou Ocaml, soit échouer.

En fait, il faut bien dire que la fréquence des erreurs de typage en pratique dans le code extrait est minime. Par exemple, il n'y en a aucune dans l'extraction de la bibliothèque standard de Coq. Et sur l'ensemble des contributions Coq étudiées du point de vue extraction (voir chapitre 5), seules 4 contributions en présentent : `Lyon/Circuits`, `Lannion`, `Rocq/Higman`, et `Nijmegen/C-CoRN`. Nous examinerons les trois dernières plus en détails dans les chapitres 5 et 6. Cette rareté s'explique par le fait que, la plupart du temps, les utilisateurs écrivent leurs fonctions informatives comme ils les auraient écrites en Ocaml ou Haskell. Ainsi les types dépendants ne sont utilisés le plus souvent que comme une forme de polymorphisme, ou encore pour exprimer des propriétés logiques (pré/post-conditions par exemple). On peut aussi voir là une forme d'auto-censure : la plupart de ces exemples datant d'avant ce remaniement de l'extraction, l'utilisation de fonctionnalités trop avancées donnait alors l'assurance de ne pas pouvoir extraire son développement.

Voyons maintenant plus en détail un certain nombre de situations typiques génératrices d'erreurs de typage pour l'extraction.

3.1.1 Le type « entier ou booléen »

Prenons par exemple un prédicat `P` dépendant d'un booléen, et valant soit `nat` soit `bool` :

```
Definition P (b:bool) : Set := if b then nat else bool.
```

Ce prédicat permet maintenant l'agglomération de deux valeurs de types différents :

```
Definition p (b:bool) : P b :=
 match b return P b with
  | true ⇒ 0
  | false ⇒ true
 end.
```

Le type de `p` est donc dépendant de `b`, car `(p true)` a pour type `(P true) = nat`, tandis que `(p false)` a pour type `(P false) = bool`. L'extraction du chapitre précédent propose alors pour `p` le terme extrait non-typable suivant :

```
let p = function
  | True → O
  | False → True
```

Il n'y a évidemment pas d'équivalent ni en Ocaml ni en Haskell d'un tel type dépendant d'un booléen. Une possibilité est alors d'oublier cette dépendance en utilisant une approximation de `P` comme étant l'union disjointe de `nat` et de `bool` :

```
type approx_P = Nat of nat | Bool of bool
let p = function
  | True → Nat O
  | False → Bool True
```

Il faudrait alors permettre également le retour vers `nat` ou `bool` une fois que le premier argument est connu. Cela passe par un repérage de chaque endroit en Coq où la règle de typage par conversion `(P true)` → `nat` est utilisée. On y insérerait la décapsulation suivante lors de l'extraction :

```
let nat_of_P = function (Nat n) → n | _ → assert false
```

Et idem avec `false` et `bool`. Une telle transformation présente plusieurs inconvénients. Déjà, elle dénature le programme initial en introduisant des encapsulations/décapsulations qui entraînent des calculs sans équivalents dans le terme original. Et d'autre part la génération automatique de ces décapsulations est loin d'être simple. En effet, l'usage de la conversion `(P true)` → `nat` est transparent pour Coq, et n'est donc enregistré nulle part. Comme on le verra par la suite, ce n'est donc pas cette méthode qui a été choisie pour rendre cet exemple.

3.1.2 Une version plus réaliste

Il est à noter que même si l'exemple précédent est simplifié au point d'en devenir un exemple-jouet, il est inspiré de développements réels. Essayons par exemple de formaliser la sémantique d'un mini-langage impératif à la Pascal, à l'aide d'un modèle-mémoire. Si notre langage ne possède que les entiers comme type de base, la fonction d'accès à une case de la mémoire va avoir un type très simple :

```
Parameter get_value : memory → address → nat.
```

avec `memory` représentant un état de la mémoire et `address` l'adresse d'une case-mémoire. Ajoutons maintenant d'autres types de base, et des références. On peut ainsi écrire par exemple :

```
Inductive types : Set :=
```

.../...

```
...
  | Nat : types
  | Bool : types
  | Address : types → types. (* type d'une référence à un autre type *)
```

Quel type donner maintenant à notre fonction `get_value` ? Bien sûr, on peut représenter les valeurs avec un type somme :

```
Inductive values : Set :=
 | Val_nat : nat → values
 | Val_bool : bool → values
 | Val_ref : address → values.

Parameter get_value : memory → address → values.
```

Le problème d'une telle représentation est qu'il faut sans arrêt raisonner par cas sur ce type `values`, même si l'on sait déjà qu'on est dans l'une des situations.

Si maintenant on utilise des types dépendants, beaucoup de ces raisonnements par cas vont devenir du simple calcul. Pour cela, associons à une case-mémoire le type de son contenu :

```
Parameter get_type : memory → address → types.
```

Et utilisons une fonction retournant le domaine associé à chaque type :

```
Definition domain (t:types) : Set :=
 match t with Nat ⇒ nat | Bool ⇒ bool | Address _ ⇒ address end.
```

On peut alors donner la signature suivante à `get_value` :

```
Parameter get_value : ∀m:memory, ∀a:address, domain (get_type m a).
```

De la sorte, si on implante la fonction `get_type` de façon réellement calculable, alors le type `(domain (get_type m a))` pourra se réduire vers `nat`, `bool` ou `address` selon le cas par simple conversion.

Une implantation de ce `get_value` présentera alors exactement le même genre de difficulté de typage lors de l'extraction que notre exemple simpliste `p`.

3.1.3 Le type des fonctions entières d'arité `n`

La situation se corse lorsque le type dépendant peut se réduire selon les cas vers une infinité de types différents. Par exemple le prédicat `F` suivant donne en fonction de l'entier `n` le type des fonctions entières à `n` arguments :

```
Fixpoint F (n:nat) : Set :=
 match n with
  | 0 ⇒ nat
  | S n ⇒ nat → F n
 end.
```

On peut alors bâtir une fonction dont l'arité dépend de son premier argument :

```
Fixpoint f (n:nat) : F n :=
 match n return F n with
  | O ⇒ O
  | S n ⇒ fun _ ⇒ f n
 end.
```

Là encore, l'extraction brute de `f` est non-typable :

```
let rec f = function
  | O → O
  | S n → (fun x → f n)
```

On peut encore proposer une version typable en utilisation un type union qui va mimer la structure de `F`, mais cela devient vraiment ardu à décrire dans le cas général d'un `Fixpoint` sur les types. Surtout qu'il est possible de cacher ce fixpoint derrière une constante. Ainsi `F` peut également s'écrire comme :

```
Definition F := nat_rect (fun _ ⇒ Set) nat (fun _ t ⇒ nat → t).
```

3.1.4 Des inductifs intraduisibles

Les inductifs posent également problème, car Coq permet d'écrire des choses sans équivalent en Ocaml ou Haskell. Par exemple, un inductif pouvant contenir des objets de n'importe quel type peut s'écrire en Coq :

```
Inductive any : Type := Any : ∀A:Type, A → any.
```

Une extraction typée naïve `type 'a any = Any of 'a` serait insatisfaisante. En effet on propagerait une variable de type `'a` sans contrepartie en Coq. Une version initiale de l'implantation de notre extraction procédait de la sorte, et l'on se retrouvait parfois avec des inductifs à 100 variables de types, à cause de cette propagation de variables.

Dans le même esprit, on peut obtenir une liste non homogène :

```
Inductive anyList : Type :=
 | AnyNil : anyList
 | AnyCons : ∀A:Type, A → anyList → anyList.

Definition my_anyList := AnyCons bool true (AnyCons nat O AnyNil).
```

Ici, une traduction naïve (qui redonnerait en fait les listes polymorphes usuelles) serait non seulement insatisfaisante, mais surtout incorrecte du point de vue typage. Cet exemple ne semble pas pouvoir être adapté vers du code Ocaml typable et équivalent. Le seul recours est alors celui présenté dans la section suivante.

Pour montrer que ces listes sont parfaitement utilisables en Coq, voici par exemple la fonction retournant la tête de la liste lorsque celle-ci est non-vide, ou bien un objet de type

unit dans le cas contraire. Le type de cette fonction utilise un prédicat dépendant de la liste :

```
Definition getHeadType (l:anyList) : Type :=
 match l with
  | AnyNil ⇒ unit
  | AnyCons A _ _ ⇒ A
 end.

Definition getHead (l:anyList) : getHeadType l :=
 match l return getHeadType l with
  | AnyNil ⇒ tt
  | AnyCons _ a _ ⇒ a
 end.
```

L'extraction de getHead présente des problèmes voisins de ceux de l'exemple initial p.

3.1.5 Types dépendants et polymorphisme

Même lorsque les types dépendants servent à exprimer un simple polymorphisme, on peut avoir de mauvaises surprises. L'exemple classique est une variante de distr_pair :

```
Definition distr_pair : (∀X:Set, X → X) → nat*bool :=
 fun f ⇒ (f nat 0, f bool true).
```

Or on sait que fun f → (f 0, f true) est non-typable. Et là encore, pas d'adaptation simple vers un code équivalent et typable.

3.1.6 Cas absurdes et typage

Un dernier exemple particulièrement déroutant est celui de l'utilisation d'une hypothèse absurde à des fins de changement de type. Supposons par exemple qu'un axiome fasse l'hypothèse que nat = bool

```
Section Strange.
 Variable absurd : nat = bool.
```

Nous pouvons alors utiliser cette fausse égalité pour montrer que 0 est un booléen.

```
 Definition 0_as_bool : bool.
  rewrite <- absurd; exact 0.
 Defined.
```

Et dès lors, rien n'interdit de définir un terme par cas sur 0 valant true ou false !

```
 Definition strange := if 0_as_bool then 0 else (S 0).
End Strange.
```

Il est à noter qu'en interne, `O_as_bool` est un simple appel à `eq_rec`, le principe d'induction sur l'égalité.

```
Coq < Print O_as_bool.
O_as_bool =
fun absurd : nat = bool ⇒ eq_rec nat (fun P : Set ⇒ P) O bool absurd
    : nat = bool → bool
```

Comme expliqué au chapitre précédent, `eq` est l'exemple emblématique des inductifs singletons logiques, il disparaît donc à l'extraction, ne laissant pour `eq_rec` que l'identité. De plus, l'argument logique `nat=bool` est ignoré et devient un `_`. L'extraction de `O_as_bool` est alors :

```
let O_as_bool _ = O
```

Quant à `strange`, il donne alors, pour peu que l'on remplace `O_as_bool` par sa définition[1] :

```
let strange _ =
 match O with
   | True → O
   | False → S O
```

Il est à noter que même si un terme extrait d'une preuve Coq peut contenir un appel à `strange`, cet appel ne sera jamais évalué, ou alors sans l'argument qui déclencherait l'évaluation du filtrage puis l'erreur d'exécution. Le corps de `strange` est en fait ici du code mort. Mais à défaut de conduire à des soucis d'exécution, il est indéniable que l'extraction de `strange` pose problème au niveau du typage.

3.2 Une correction artificielle des erreurs de typage

3.2.1 Obj.magic et unsafeCoerce

Nous avons vu que certaines de ces erreurs de typage pouvaient éventuellement se corriger par remaniement du code. Mais toutes ne se prêtent pas à ce processus, qui est de toute façon difficilement automatisable et en outre non souhaitable. C'est pourquoi nous avons suivi une approche uniforme consistant à utiliser des primitives bas-niveau et non-documentées qui autorisent à changer artificiellement le type d'une expression. Par exemple, pour Ocaml, il s'agit de `Obj.magic : 'a → 'b`. Une telle primitive, implantée en interne par l'identité, n'est évidemment pas définissable normalement en Ocaml, et son usage fait sortir du cadre de la plupart des résultats théoriques : en particulier, le fait pour un terme Ocaml avec `Obj.magic` d'être bien typé ne suffit plus à garantir son exécution sans erreur.

Certaines implantations de Haskell présentent une primitive nommée `unsafeCoerce`, qui devrait pouvoir s'utiliser exactement de la même façon que `Obj.magic`. Faute de temps, nous n'avons pas implanté la génération automatique de `unsafeCoerce` dans le code Haskell

[1] Voir plus tard en section 4.3.3 la discussion sur l'opportunité de tels remplacements.

extrait. Mais cette génération ne pose, a priori, pas plus de problème que la génération des `Obj.magic`. La suite de ce chapitre est donc consacrée à Ocaml.

Faisons par exemple passer un booléen pour un entier grâce un `Obj.magic`.

```
# (Obj.magic true) + 1;;
- : int = 2
```

Le fait que le calcul réussisse est ici une conséquence de la représentation interne des objets en Ocaml : `true` étant codé par 1, on obtient 2 au final. Par contre, si la chimère créée à coup de `Obj.magic` n'est pas compatible avec la représentation interne des objets, la sanction est immédiate :

```
# (Obj.magic 1) 2;;
Segmentation fault
```

En effet, on essaie d'utiliser ici 1 comme pointeur vers le code d'une fonction, ce qui engendre une erreur mémoire.

L'usage des `Obj.magic` par le programmeur, sans être interdit, est donc en tout cas fortement déconseillé, et doit être assorti de la plus grande prudence. Mais on ne doit pas non plus diaboliser ces `Obj.magic`, qui dans certaines situations sont bien pratiques. Par exemple on en retrouve dans l'implantation actuelle du module Queue de la bibliothèque standard de Ocaml, ou dans un module Dyn de typage dynamique dans les sources de Coq.

Dans le cadre de l'extraction, heureusement, nous savons déjà qu'une erreur d'exécution n'est pas possible. Nous avons alors la liberté de placer autant de `Obj.magic` que nécessaire afin d'assurer le typage. Une version corrigée de l'extraction de l'exemple `p` de la section 3.1.1 peut par exemple être :

```
let p = function
  | True → Obj.magic 0
  | False → Obj.magic True
```

Comment placer alors ces `Obj.magic` ? Une possibilité est évidemment d'en mettre à chaque noeud du programme. Mais cela présente deux inconvénients :

- Le code devient complètement illisible, ce qui va à l'encontre d'un de nos objectifs.
- Plus grave, les performances sont diminuées, comme l'a montré L. Pottier [70]. Ceci s'explique (au moins en partie) par le fait que le compilateur Ocaml désactive un certain nombre d'optimisations autour des `Obj.magic`.

Comme on a vu que les erreurs de typages restent le plus souvent très marginales dans le code extrait, il est donc réellement gagnant de les repérer finement afin d'insérer le moins de `Obj.magic` possible.

3.2.2 Une première tentative de correcteur de typage

Nous avons donc embarqué dans notre extraction un vérificateur de typage faisant également fonction de correcteur de typage : à chaque erreur détectée, on insère un `Obj.magic` la

faisant disparaître. Nous allons voir une manière relativement simple, mais aussi peu satisfaisante de faire cela. Puis nous décrirons ensuite la méthode réellement utilisée en pratique dans l'extraction.

Les algorithmes $\mathcal{W}$ et $\mathcal{W}$'

La façon la plus simple de détecter et corriger les erreurs de type est de procéder de manière *paresseuse*. Pour cela on utilise un algorithme d'inférence/vérification de type, comme par exemple l'algorithme $\mathcal{W}$ de Damas-Milner [27]. Le type de l'algorithme est le suivant :

```
W : env * expr → type * subst
```

Et sa correction s'exprime par $\mathcal{W}(\Gamma, t) = (T, \sigma) \;\Rightarrow\; \sigma(\Gamma) \vdash t : \sigma(T)$.

Cet algorithme ne peut échouer à typer un terme que via un échec de l'un des appels à l'unificateur de types `mgu`. Tout ce qui nous reste à faire consiste à rattraper ces erreurs d'unification et à les transformer en succès grâce à des `Obj.magic`. Pour cela nous adaptons $\mathcal{W}$ afin qu'il retourne également les modifications apportées aux termes. Son type va maintenant être :

```
W' : env * expr → type * subst * expr
```

Et la propriété souhaitée est maintenant $\mathcal{W}'(\Gamma, t) = (T, \sigma, t') \;\Rightarrow\; \sigma(\Gamma) \vdash t' : \sigma(T)$ et $t' \sim t$.

L'algorithme $\mathcal{W}$ étant désormais plus que classique, nous allons donner ici une présentation de $\mathcal{W}$' axée sur nos modifications. Une présentation plus formelle peut être trouvée dans [51]. La présentation donnée ici s'inspire d'un cours de X. Leroy.

Tout d'abord, rappelons juste les mécanismes d'instanciation et de généralisation permettant de passer d'un schéma de type Ocaml à un type et réciproquement :

`Inst`$(\forall \overrightarrow{\alpha_i}.\tau) = \tau[\overrightarrow{\alpha_i}/\overrightarrow{\beta_i}]$, avec $\overrightarrow{\beta_i}$ variables fraîches.

`Gen`$(\tau, \Gamma) = \forall \overrightarrow{\alpha_i}.\tau$, avec $\overrightarrow{\alpha_i}$ les variables de τ libres dans Γ

La figure 3.1 contient la définition d'une version minimaliste de $\mathcal{W}$', paramétrée par un environnement E de déclarations de constantes munies de leurs types. Cette présentation ne comporte volontairement pas de récursivité : au niveau du typage, on peut la présenter via un opérateur de point-fixe `fix` ayant pour signature $\forall\alpha,\ (\alpha \to \alpha) \to \alpha$.

L'encodage des inductifs

Les inductifs vont aussi être représentés par encodage via des constantes, afin de laisser l'algorithme central aussi simple que possible. Prenons un type inductif Ocaml t défini par :

`type` $(\alpha_1, \ldots, \alpha_2)\ t = C_1$ `of` $\tau_{1,1} * \ldots * \tau_{1,n_1} | \ldots |\ C_p$ `of` $\tau_{p,1} * \ldots * \tau_{p,n_p}$

On représente les constructeurs de t par des constantes $C_1 \ldots C_p$ ayant pour types :

$C_i \;:\; \forall\alpha_1, \ldots, \alpha_n.\tau_{i,1} \to \ldots \to \tau_{i,n_i} \to (\alpha_1, \ldots, \alpha_n)\ t$

Quant au filtrage sur t, on le représente par un opérateur F_t du type suivant :

$$F_t \;:\; \forall\alpha_1, \ldots, \alpha_n, \beta.(\alpha_1, \ldots, \alpha_n)\ t \to (\tau_{1,1} \to \ldots \to \tau_{1,n_1} \to \beta) \to \ldots \to (\tau_{p,1} \to \ldots \to \tau_{p,n_p} \to \beta) \to \beta$$

$\mathcal{W}'(\Gamma,\ x) = (\texttt{Inst}(\Gamma(x)),\ \texttt{id},\ x)$ pour une variable x

$\mathcal{W}'(\Gamma,\ c) = (\texttt{Inst}(E(x)),\ \texttt{id},\ c)$ pour une constante c

$\mathcal{W}'(\Gamma,\ \texttt{fun}\ x \to a) =$
$\quad \texttt{let}\ (\tau_1,\ \phi_1,\ \tilde{a}) = \mathcal{W}'(\Gamma + \{x : \beta\},\ a)\ \texttt{in}$ avec β variable fraîche
$\quad (\phi_1(\beta) \to \tau_1,\ \phi_1,\ \texttt{fun}\ x\ \to \tilde{a})$

$\mathcal{W}'(\Gamma,\ a_1\ a_2) =$
$\quad \texttt{let}\ (\tau_1,\ \phi_1,\ \tilde{a_1}) = \mathcal{W}'(\Gamma,\ a_1)\ \texttt{in}$
$\quad \texttt{let}\ (\tau_2,\ \phi_2,\ \tilde{a_2}) = \mathcal{W}'(\phi_1(\Gamma),\ a_2)\ \texttt{in}$
$\quad \texttt{let}\ \sigma = \texttt{mgu}(\phi_2(\tau_1),\ \alpha \to \beta)\ \texttt{in}$ avec α et β fraîches
$\quad \texttt{if}\ \sigma = \texttt{error then}$
$\quad\quad (\gamma,\ \phi_2 \cdot \phi_1,\ (\texttt{Obj.magic}\ \tilde{a_1})\ \tilde{a_2})$ avec γ fraîche
$\quad \texttt{else}$
$\quad\quad \texttt{let}\ \mu = \texttt{mgu}(\tau_2,\ \sigma(\alpha))\ \texttt{in}$
$\quad\quad \texttt{if}\ \mu = \texttt{error then}$
$\quad\quad\quad (\sigma(\beta),\ \ \phi \cdot \phi_2 \cdot \phi_1,\ \ \tilde{a_1}\ (\texttt{Obj.magic}\ \tilde{a_2}))$
$\quad\quad \texttt{else}$
$\quad\quad\quad (\mu(\beta),\ \ \mu \cdot \sigma \cdot \phi_2 \cdot \phi_1,\ \ \tilde{a_1}\ \tilde{a_2})$

$\mathcal{W}'(\Gamma,\ \texttt{let}\ x = a_1\ \texttt{in}\ a_2) =$
$\quad \texttt{let}\ (\tau_1,\ \phi_1,\ \tilde{a_1}) = \mathcal{W}'(\Gamma,\ a_1)\ \texttt{in}$
$\quad \texttt{let}\ (\tau_2,\ \phi_2,\ \tilde{a_2}) = \mathcal{W}'(\phi_1(\Gamma) + \{x : \texttt{Gen}(\tau_1,\ \phi_1(\Gamma))\},\ a_2)\ \texttt{in}$
$\quad (\tau_2,\ \phi_2 \cdot \phi_1,\ \texttt{let}\ x = \tilde{a_1}\ \texttt{in}\ \tilde{a_2})$

FIG. 3.1: Définition de $\mathcal{W}$'

Une simplification à éviter

À première vue, notre algorithme $\mathcal{W}$' semble pouvoir être simplifié au niveau de la génération des `Obj.magic`. En effet, pour une application $(a_1\ a_2)$, on distingue deux cas :

- si la tête a_1 n'a pas un type flèche, on l'entoure d'un `Obj.magic`.
- si l'argument a_2 n'a pas un type compatible avec le début du type flèche, c'est cet argument qui reçoit un `Obj.magic`.

Si un `Obj.magic` est nécessaire, une solution également correcte et plus simple consiste à toujours le mettre autour de la tête de l'application. Le problème est que les constantes C_i et F_t encodant des inductifs peuvent alors se retrouver entourées par des `Obj.magic`, ce qui empêche de les retransformer en syntaxe Ocaml. Notre solution avec deux unifications évite ce problème.

Correction de $\mathcal{W}$'

Cet algorithme $\mathcal{W}$' vérifie bien la propriété $\mathcal{W}'(\Gamma, t) = (T, \sigma, t') \Rightarrow \sigma(\Gamma) \vdash t' : \sigma(T)$. La preuve se fait de la même façon que celle de l'algorithme non modifié $\mathcal{W}$. Il faut juste vérifier en plus les deux cas de génération de `Obj.magic`. Or on a par hypothèse de récurrence $\phi_1(\Gamma) \vdash \tilde{a_1} : \tau_1$ et $\phi_2(\phi_1(\Gamma)) \vdash \tilde{a_2} : \tau_2$. De la première relation on obtient $\phi_2(\phi_1(\Gamma)) \vdash \tilde{a_1} : \phi_2(\tau_1)$.

- Le premier cas de génération de `Obj.magic` est le plus simple : le schéma de type $\forall\alpha\beta.\alpha \to \beta$ permet de donner à (`Obj.magic` $\tilde{a_1}$) n'importe quel type, y compris $\tau_2 \to \gamma$. L'application ((`Obj.magic` $\tilde{a_1}$) $\tilde{a_2}$) accepte donc bien γ comme type dans l'environnement $\phi_2(\phi_1(\Gamma))$
- Dans le deuxième cas, on a $\sigma(\phi_2(\phi_1(\Gamma))) \vdash \tilde{a_1} : \sigma(\phi_2(\tau_1))$. Compte tenu de la définition de σ, ceci se réécrit en $\sigma(\phi_2(phi_1(\Gamma))) \vdash \tilde{a_1} : \sigma(\alpha) \to \sigma(\beta)$. C'est maintenant (`Obj.magic` $\tilde{a_2}$) qui peut prendre n'importe quel type et en particulier $\sigma(\alpha)$, ce qui fait au final que ($\tilde{a_1}$ (`Obj.magic` $\tilde{a_2}$) accepte donc bien $\sigma(\beta)$ comme type dans l'environnement $\sigma(\phi_2(\phi_1(\Gamma)))$.

Fonctionnement de $\mathcal{W}$' sur un exemple

Testons maintenant notre algorithme $\mathcal{W}$' sur l'extraction de l'exemple `p`. En utilisant notre encodage, le corps de `p` s'écrit : `fun` $b \to F_{bool}$ b `O True`. Les appels successifs à $\mathcal{W}$' vont alors être, du plus interne au plus externe :

$\mathcal{W}'(\{b : \alpha\}, F_{bool}) = (\texttt{bool} \to \beta \to \beta \to \beta,\ \texttt{id},\ F_{bool})$
$\mathcal{W}'(\{b : \alpha\}, F_{bool}\ b) = (\beta \to \beta \to \beta,\ \{\alpha \leftarrow \texttt{bool}\},\ F_{bool}\ b)$
$\mathcal{W}'(\{b : \alpha\}, F_{bool}\ b\ \texttt{O}) = (\texttt{nat} \to \texttt{nat},\ \{\alpha \leftarrow \texttt{bool}; \beta \leftarrow \texttt{nat}\},\ F_{bool}\ b\ \texttt{O})$
$\mathcal{W}'(\{b : \alpha\}, F_{bool}\ b\ \texttt{O True}) = (\texttt{nat},\ \{\alpha \leftarrow \texttt{bool}; \beta \leftarrow \texttt{nat}\},\ F_{bool}\ b\ \texttt{O (Obj.magic True)})$
$\mathcal{W}'(\varnothing, \texttt{fun}\ b \to F_{bool}\ b\ \texttt{O True}) = (\texttt{bool} \to \texttt{nat}, \ldots, \texttt{fun}\ b \to F_{bool}\ b\ \texttt{O (Obj.magic True)})$

À la quatrième ligne, `True` ne peut avoir le type `nat` nécessaire à une application correcte. On l'entoure donc d'un `Obj.magic`. Au final, le terme corrigé est donc (dans la syntaxe Ocaml) :

```
let p = function
  | True → O
  | False → Obj.magic True
```

Les limitations de $\mathcal{W}$'

Si cette méthode marche donc, et produit bien des termes typables, elle est néanmoins peu satisfaisante. En effet les types Ocaml inférés sont parfois étranges et dissymétriques, l'ordre d'unification jouant un rôle important dans le résultat final. Ainsi pour notre exemple `p` le problème de typage n'est découvert que sur la deuxième branche du `match`, et c'est donc seulement cette deuxième branche qui porte un `Obj.magic`. Ceci fait que le type de `p` est

`bool→ nat`, et que les occurrences de `(p false)` seront entourées de `Obj.magic`, alors que les occurrences de `(p true)` ne le seront pas.

Cette dissymétrie empêche de prévoir a priori quel va être le type inféré pour un terme extrait donné. La seule façon de calculer ce type est d'appliquer l'algorithme $\mathcal{W}$'. En particulier deux termes Coq de même type n'auront pas forcément le même type Ocaml une fois extraits. Pour s'en convaincre, il suffit de considérer `p` et sa variante suivante, où l'on filtre dans l'autre sens en utilisant le booléen opposé :

```
Definition p' :=
 fun b: bool ⇒
 match negb b as b' return P (negb b') with
  | true ⇒ true
  | false ⇒ O
 end.
```

L'extraction via $\mathcal{W}$' de `p'` donne alors :

```
let p' b =
  match negb b with
    | True → Obj.magic True
    | False → O
```

Et ce terme extrait a pour type `bool→bool` au lieu du `bool→nat` de `p`. Il faut ici préciser un léger abus. En effet le type Coq de `p'` est `∀b:bool, P (negb (negb b))`, qui n'est pas exactement celui de `p`, qui est `∀b:bool, P b`. Pour corriger cet abus, il suffit par exemple de considérer non pas `p` et `p'`, mais plutôt `(p true)` et `(p' true)`. Ces deux derniers termes ont tous les deux le type `nat` puisque `(P true) = (P (negb (negb true))) = nat`. Par contre après extraction `(p true)` et `(p' true)` vont avoir pour types Ocaml respectifs `nat` et `bool`[2].

Cette impossibilité de prévoir à l'avance les types des termes extraits exclut l'utilisation de cette méthode. En effet, ceci est très gênant en cas d'interfaçage de code extrait dans un développement plus large. Et même sans parler d'intégration avec du code externe, l'extraction des modules Coq vers des modules Ocaml (cf. section 4.1) est fortement compliquée par une telle propriété.

3.2.3 L'algorithme $\mathcal{M}$

Pour remédier à ces difficultés, nous allons procéder en deux phases.

- La première phase sera le calcul d'un type Ocaml $\widehat{\mathcal{E}}(T)$ attendu pour le terme extrait à partir de $t : T$. Ce calcul se fait indépendamment du corps du terme Coq t, et n'utilise

[2]Une autre façon d'obtenir deux termes adéquats est d'enrober `p'` en un terme `p''` ayant le même type que `p`. En effet les types de `p` et `p'` étant prouvablement égaux à défaut d'être convertibles, on peut passer de l'un à l'autre via le lemme `negb_elim : ∀b:bool, negb (negb b)=b`. Ceci nous donne :

```
Definition p'' : ∀b :bool, P b. intro b ; rewrite <- negb_elim ; exact (p' b). Defined.
```

que son type T, afin de rester modulaire. Cette phase d'extraction $\widehat{\mathcal{E}}$ sera détaillée ultérieurement.

- Une fois muni de ce type qui va nous servir d'objectif, nous pouvons contraindre l'extraction de t à effectivement accepter $\widehat{\mathcal{E}}(T)$ comme type Ocaml, toujours grâce à des `Obj.magic`.

Nous allons commencer par détailler cette seconde phase, qui va se faire encore une fois en adaptant un algorithme de typage. Mais désormais l'accent va davantage être mis sur la vérification de type que sur l'inférence. Et à ce changement d'approche va correspondre un changement d'algorithme : nous allons remplacer $\mathcal{W}$ par l'algorithme $\mathcal{M}$. Cet algorithme, décrit en 1998 par O. Lee et K. Yi dans [51], est lui-même une variante de $\mathcal{W}$. Mais à la différence de ce dernier, $\mathcal{M}$ procède par analyse descendante et non ascendante, d'où son nom renversé. Pour la petite histoire, cet algorithme $\mathcal{M}$ a été utilisé jusqu'en 1993 dans les versions de Caml Light avant la version 0.7. Et même si nous ne tirerons pas partie de cette propriété, on peut néanmoins noter que $\mathcal{M}$ détecte les erreurs de typage de manière plus fine que $\mathcal{W}$. Le lecteur intéressé par plus de détails à ce propos pourra se reporter à [51].

La différence entre $\mathcal{M}$ et $\mathcal{W}$ se voit déjà au niveau de leur types :

```
W : env * expr → type * subst
M : env * expr * type → subst
```

Au lieu d'inférer un type résultat, $\mathcal{M}$ part plutôt d'un type en entrée et vérifie que ce type peut convenir, modulo une éventuelle substitution. Sa correction s'exprime par :

$\mathcal{M}(\Gamma, t, T) = \sigma \;\Rightarrow\; \sigma(\Gamma) \vdash t : \sigma(T)$.

En pratique, la différence de nature entre les deux algorithmes n'est pas si grande, il suffit, en effet, d'utiliser $\mathcal{M}$ avec un type initial `'x` pour retrouver un algorithme d'inférence, et le type inféré obtenu à la fin est $\sigma($`'x`$)$. Et de toute façon, même un algorithme axé sur la vérification comme $\mathcal{M}$ doit faire des étapes d'inférence, par exemple lorsqu'il rencontre un `let in`.

Pour utiliser $\mathcal{M}$ dans le cadre de l'extraction, on l'adapte de la même façon que l'on avait adapté $\mathcal{W}$ en $\mathcal{W}$' :

```
M' : env * expr * type → subst * expr
```

Et l'on désire maintenant que $\mathcal{M}'(\Gamma, t, T) = (\sigma, t') \;\Rightarrow\; \sigma(\Gamma) \vdash t' : \sigma(T)$. La figure 3.2 donne la définition de $\mathcal{M}$', qui utilise les mêmes notations que la définition de $\mathcal{W}$'. On reprend également le même encodage de la récursivité et des inductifs.

Si l'on compare $\mathcal{M}$' et $\mathcal{W}$', on remarque que le cas du `let in` est similaire dans les deux algorithmes. Ceci s'explique par le besoin d'inférer le type de la définition locale. Pour les autres cas, les rôles sont renversés concernant l'usage de l'unificateur `mgu`. L'application ne nécessite plus d'unification, alors qu'à l'opposé les derniers cas (fonctions, constantes et variables) en contiennent maintenant.

La correction de $\mathcal{M}$' ne sera pas développée. Tout comme pour $\mathcal{W}$ et $\mathcal{W}$', cette correction repose sur celle de $\mathcal{W}$, dont on peut trouver la preuve dans [51], et sur l'étude des cas d'ajout de `Obj.magic`.

$\mathcal{M}'(\Gamma,\ x,\ \tau) =$
  `let` $\sigma = \texttt{mgu}(\tau,\ \texttt{Inst}(\Gamma(x)))$ `in`
  `if` $\sigma = \texttt{error}$ `then` $(\texttt{id},\ \texttt{Obj.magic}\ x)$ `else` $(\sigma,\ x)$

$\mathcal{M}'(\Gamma,\ c,\ \tau) =$
  `let` $\sigma = \texttt{mgu}(\tau,\ \texttt{Inst}(E(c)))$ `in`
  `if` $\sigma = \texttt{error}$ `then` $(\texttt{id},\ \texttt{Obj.magic}\ c)$ `else` $(\sigma,\ c)$

$\mathcal{M}'(\Gamma,\ \texttt{fun}\ x \to a,\ \tau) =$
  `let` $\sigma = \texttt{mgu}(\tau,\ \alpha \to \beta)$ `in` avec α et β fraîches
  `if` $\sigma = \texttt{error}$ `then`
    `let` $(\phi,\ \tilde{a}) = \mathcal{M}'(\Gamma + \{x : \alpha\},\ a,\ \beta)$ `in`
    $(\phi,\ \texttt{Obj.magic}\ (\texttt{fun}\ x \to \tilde{a}))$
  `else`
    `let` $(\phi,\ \tilde{a}) = \mathcal{M}'(\Gamma + \{x : \sigma(\alpha)\},\ a,\ \sigma(\beta))$ `in`
    $(\phi \cdot \sigma,\ \texttt{fun}\ x \to \tilde{a})$

$\mathcal{M}'(\Gamma,\ a_1\ a_2) =$
  `let` $(\phi_1,\ \tilde{a_1}) = \mathcal{M}'(\Gamma,\ a_1,\ \alpha \to \tau)$ `in` avec α fraîche
  `let` $(\phi_2,\ \tilde{a_2}) = \mathcal{M}'(\phi_1(\Gamma),\ a_2,\ \phi_1(\alpha))$ `in`
  $(\phi_2 \cdot \phi_1,\ \tilde{a_1}\ \tilde{a_2})$

$\mathcal{M}'(\Gamma,\ \texttt{let}\ x = a_1\ \texttt{in}\ a_2) =$
  `let` $(\phi_1,\ \tilde{a_1}) = \mathcal{M}'(\Gamma,\ a_1,\ \alpha)$ `in` avec α fraîche
  `let` $(\phi_2,\ \tilde{a_2}) = \mathcal{M}'(\phi_1(\Gamma) + \{x : \texttt{Gen}(\phi_1(\alpha),\ \phi_1(\Gamma))\},\ a_2,\ \phi_1(\tau))$ `in`
  $(\phi_2 \cdot \phi_1,\ \texttt{let}\ x = \tilde{a_1}\ \texttt{in}\ \tilde{a_2})$

FIG. 3.2: Définition de $\mathcal{M}$'

Mais nous ne pouvons toujours pas utiliser $\mathcal{M}$' directement dans le cadre de l'extraction. En effet on désire pour un type τ donné, forcer un terme a à accepter *exactement* τ comme type. Or $\mathcal{M}'$ nous retourne une substitution σ à appliquer sur les variables de τ pour que τ convienne comme type de a, ce qui fait que le type final $\sigma(\tau)$ de a peut être strictement moins général que τ. Pour résoudre ce problème, nous avons utilisé deux types de variables :

- les variables substituables, qui sont celles utilisées jusqu'à maintenant, mais qui ne vont plus servir que de façon interne à $\mathcal{M}$', lors des créations de variables fraîches.
- les variables non substituables, les seules autorisées dans les types extraits arguments initiaux de $\mathcal{M}$'. Si l'unificateur `mgu` doit résoudre une équation de la forme $\alpha =? \tau$ avec α variable non substituable et τ type différent de α, une erreur d'unification est levée au lieu de retourner la substitution $\{\alpha \leftarrow \tau\}$. Au vu de cette erreur d'unification, $\mathcal{M}$' va alors générer un `Obj.magic` qui va permettre de garder le type le plus général, ici α.

Désormais, en assurant que le type τ fourni à $\mathcal{M}$' ne contient que des variables substituables, on garantit donc que la substitution résultat σ laisse τ invariant. Nous allons donc pouvoir ignorer cette substitution. Au final, à partir d'un terme Coq $t : T$, le terme extrait typable t_{ml} et son type T_{ml} s'obtiennent par : $T_{ml} = \widehat{\mathcal{E}}(T)$ et $t_{ml} = \texttt{snd}(\mathcal{M}'(\varnothing, \mathcal{E}(t), \widehat{\mathcal{E}}(T)))$. Et quel que soit le choix de $\widehat{\mathcal{E}}(T)$, on est certain que t_{ml} a bien pour type T_{ml} et ne diffère de $\mathcal{E}(t)$ que par l'insertion éventuelle de `Obj.magic`.

3.3 L'extraction des types Coq

Nous allons maintenant présenter l'extraction $\widehat{\mathcal{E}}$ des types Coq vers des types à la Ocaml. Cette extraction des types, on l'a vu, est utilisée par notre algorithme $\mathcal{M}$'. Mais elle est de toute façon rendue nécessaire par la présence des inductifs en Coq : à partir des types Coq des constructeurs de l'inductif, nous devons déduire des types Ocaml les plus fidèles possibles pour les constructeurs extraits. Au niveau de l'exécution, la seule contrainte concernant les inductifs est de conserver le nombre de leurs constructeurs et le nombre d'arguments de ces constructeurs. Mais s'en tenir juste à cela obligerait à mettre beaucoup trop de `Obj.magic`, et produirait des termes et types extraits illisibles. En outre, cette extraction des types va nous permettre de produire un fichier d'interface `.mli` pour chaque fichier `.ml` produit, et cela de manière prévisible : seuls les types Coq vont influer sur cette interface, et pas le contenu des termes à extraire.

3.3.1 Une approximation des types Coq

Les exemples précédents ont montré que la richesse des types Coq ne pouvait pas être rendue en Ocaml de manière toujours fidèle. Nous procédons alors par approximation. Pour cela, nous utilisons un type Ocaml le plus général (ou encore le plus inconnu), que nous noterons $\mathbb{T}$. Nous avons implanté[3] ce type $\mathbb{T}$ grâce au type interne Ocaml de tous les objets, `Obj.t`. Et les conversions entre `Obj.t` et les autres types sont effectuées grâce à des `Obj.magic`.

Ce type $\mathbb{T}$ va permettre de donner une réponse satisfaisante aux exemples précédents. Ainsi la fonction `p` qui retourne un entier ou un booléen aura pour type `bool` $\rightarrow \mathbb{T}$. Quant à l'inductif `any` contenant n'importe quel objet, il devient :

```
type any = Any of 𝕋
```

Il doit alors être clair que l'extraction des types que nous proposons est nécessairement arbitraire à certains endroits, même si elle donne de bons résultats en pratique. Cette « efficacité » de l'extraction des types se mesure :

- à la précision de l'approximation effectuée. Une extraction des types répondant toujours $\mathbb{T}$, bien que possible, n'est guère intéressante.

[3]Dans le code extrait, la notation ASCII pour ce type $\mathbb{T}$ est `__`. A noter l'absence de conflit avec la notation du terme $\square$ qui est également `__` : en Ocaml les noms de types et de termes n'interfèrent pas.

- au faible nombre de `Obj.magic` nécessaires pour réconcilier termes extraits et types extraits.

En fait, la définition de $\widehat{\mathcal{E}}$ qui va suivre ne procède que par remplacement de sous-termes par $\mathbb{T}$, si l'on met de côté les questions de syntaxe. Il serait alors possible de munir l'ensemble de ces types pouvant contenir $\mathbb{T}$ d'une structure de semi-treillis supérieur, dont $\mathbb{T}$ serait l'élément maximal, et qui pour tout type Coq U, vérifierait $U \leq \widehat{\mathcal{E}}(U) \leq \mathbb{T}$. Cet ordre constituerait alors une mesure du degré d'approximation commise lors de l'extraction des types.

3.3.2 Le type des résidus logiques

Nous avons vu au chapitre précédent que l'extraction des termes n'était pas en mesure de faire complètement disparaître les parties logiques des termes, et qu'il pouvait subsister des constantes résiduelles $\Box$. Quel type pouvons-nous donner à ces résidus $\Box$? Idéalement n'importe quel type de constante à au moins une valeur pourrait convenir, comme par exemple `unit`. Malheureusement, il peut arriver lors d'une réduction en Ocaml qu'un résidu se retrouve appliqué (voir exemple 1 du chapitre précédent). La règle de réduction à utiliser est alors :

$$(\Box\ x) \rightarrow \Box$$

Implanter $\Box$ par `()` est donc incorrect du point de vue de l'exécution. On peut néanmoins reproduire le bon comportement en Ocaml, à condition de tricher avec le typage[4] :

```
# let rec __ x = Obj.magic __;;
val __ : 'a → 'b = <fun>
# __ 1 true [];;
- : '_a = <poly>
```

La seule difficulté est que le type `'a→'b` fourni pour $\Box$ est néfaste pour la lisibilité des signatures, car il multiplie les variables de types inutiles. Nous avons alors décidé de « caster » ce type vers `Obj.t`. Ceci peut se faire en utilisant au lieu de `Obj.magic` sa variante `Obj.repr : 'a → Obj.t`. Nous avons au final choisi l'implantation suivante de $\Box$:

```
# let __ = let rec f _ = Obj.repr f in Obj.repr f
val __ : Obj.t = <fun>
```

Dans la définition de l'extraction $\widehat{\mathcal{E}}$ des types qui va suivre, nous utiliserons le symbole $\Box$ pour désigner le type de la constante $\Box$. Ce symbole de type est défini comme un alias de $\mathbb{T}$. En fait, on ne sépare syntaxiquement ces deux symboles que le temps de l'extraction, pendant laquelle ils vont jouer des rôles distincts :

- Si l'extraction d'un type donne `nat→`$\Box$, on sait alors que tout le type est logique, et il est souhaitable de plutôt produire l'extraction $\Box$ pour le tout.

[4] La notation ASCII de $\Box$ est `__`.

- A l'inverse si un type s'extrait en `nat`$\rightarrow\mathbb{T}$, on sait qu'il s'agit d'un type de fonction à argument entier, et de résultat inconnu. Et là une approximation du tout en $\mathbb{T}$ n'est pas souhaitable car on perd alors toute information.

3.3.3 La frontière entre types et termes

Rappelons que Coq ne fait pas de distinction syntaxique entre termes et types, au contraire de Ocaml. C'est donc à l'extraction de savoir orienter un objet Coq vers le monde des termes extraits ou celui des types extraits. Dans le chapitre précédent sur l'extraction des termes, nous avons fusionné tous les schémas de types[5] vers la constante $\Box$. La raison de cette fusion était qu'un schéma de type, qui va devenir un type une fois appliqué à suffisamment d'arguments, n'a donc pas de réel contenu calculatoire.

De façon cohérente avec l'extraction des termes, c'est encore une fois les schémas de types qui vont servir de frontière entre types et termes. Ainsi l'extraction des types va devoir accepter tout terme Coq, mais va immédiatement rendre le type inconnu $\mathbb{T}$ si le terme d'entrée n'est pas un schéma de type, de la même façon que l'extraction des termes retourne $\Box$ dans les cas dégénérés.

3.3.4 Les types Coq, du simple vers le complexe

Avant d'entrer dans la définition proprement dite de l'extraction des types, nous allons tenter de donner une intuition de son comportement. Pour cela, nous allons présenter un certain nombre d'exemples de types Coq suffisamment simples pour avoir un équivalent fidèle en Ocaml. Puis nous irons vers les types Coq de plus en plus complexes.

Les inductifs

On peut évidemment mentionner tout d'abord les types inductifs dont les constructeurs sont constants, tels que `bool`.

```
Inductive bool : Set := true : bool | false : bool.
```

$\Downarrow$

```
type bool = True | False.
```

Plus généralement les types inductifs sans paramètre dont les constructeurs ont des types simplement traduisibles sont eux-mêmes simplement traduisibles. Ainsi :

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

$\Downarrow$

```
type nat = O | S of nat.
```

[5]Pour mémoire, un schéma de types est un terme admettant un type de la forme : $\forall x_1 : X_1, ...\forall x_n : X_n, s$

Si maintenant on introduit des paramètres de type `Set` (ou `Type`), la situation reste naturelle. On peut en effet voir `Set` comme l'ensemble de tous les types. Le paramètre Coq devient alors une variable de type Ocaml. Par exemple :

```
Inductive list (A:Set) : Set :=
 | nil : list A
 | cons : A → list A → list A.
```

⇓

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

A noter, le passage du type du constructeur d'une forme fonctionnelle curryfiée en Coq à un suite de produits en Ocaml. Et l'application d'un tel inductif se traduit évidemment en une application de type Ocaml : une `(list nat)` de Coq devient une `(nat list)` dans Ocaml.

Il n'y a pas de raison de se limiter aux paramètres, les produits présents dans la signature de l'inductif jouant un rôle similaire.

```
Inductive list2 : ∀A:Set, Type :=
 | nil2 : ∀A:Set, list2 A
 | cons2 : ∀A:Set, A → list2 (A*A) → list2 A.
```

⇓

```
type 'a list2 = Nil2 | Cons2 of 'a * ('a * 'a) list2
```

Mais on entre là dans une zone dangereuse. Il suffit de changer légèrement la précédente définition pour être bloqué.

```
Inductive list3 : ∀A:Set, Type :=
 | nil3 : ∀A:Set, list3 A
 | cons3 : ∀A:Set, A → list3 A → list3 (A*A).
```

Il s'agit du dual du précédent, dans lequel les pairs vont s'accumuler vers la droite des listes. Mais en Ocaml, pour un inductif `list3` à un argument, les constructeurs servent à construire un `'a list3` implicite. Il n'y a donc pas moyen de traduire fidèlement cet inductif, et on devra donc choisir une approximation. Par exemple

```
type 'a list3 = Nil3 | Cons3 of 𝕋 * 𝕋 list3
```

Avec ces paramètres ou variables dans le type `Set`, on a déjà atteint l'ensemble des types inductifs définissables en Ocaml. Tout type inductif Coq plus complexe va donc amener des soucis. Si par exemple un paramètre est un schéma de type, le représenter par une variable de type va être imparfait : cette variable, au lieu de représenter toute une famille potentielle de types, ne va en représenter qu'un. Nous prendrons néanmoins cette représentation, car on ne peut faire mieux en Ocaml. Par exemple :

```
Inductive poly (X:nat→Set) : Set := Poly : ∀n:nat, X n → poly X.
```

⇓

```
type 'x poly = Poly of nat * 'x
```

Enfin, dernier cas, si un paramètre ou une variable est au niveau des termes, il ne sera pas traduisible en Ocaml. L'exemple classique est celui des listes de taille n.

```
Inductive listn (A:Set) : nat → Set :=
 | niln : listn A O
 | consn : ∀n:nat, A → listn A n → listn A (S n).
```

Une solution est alors de remplacer chaque argument intraduisible par $\mathbb{T}$.

```
type ('a,'n) listn = Niln | Consn of nat * 'a * ('a,𝕋) listn
```

Mais nous verrons plus tard que l'on peut toujours repérer de tel dépendances vis-à-vis de termes et les faire complètement disparaître.

Les schémas de types

Les schémas de types vont correspondre assez naturellement aux types Ocaml avec variables de type. Prenons par exemple le cas d'un schéma de type ayant une dépendance par rapport à un type :

```
Definition Sch1 : Set → Set := fun X:Set ⇒ X → X.
```

⇓

```
type 'x sch1 = 'x → 'x
```

Et si l'on applique ce schéma de types en Coq, on obtient une application également en Ocaml : `(Sch1 nat)` donne `(nat sch1)` en syntaxe Ocaml.

Bref, comme pour les inductifs, tant que la dépendance est par rapport à un type, tout reste simple. Considérons maintenant le cas d'une dépendance vis-à-vis d'un terme. Il suffit par exemple de reprendre la constante `P` de la section 3.1 :

```
Definition P (b:bool) : Set := if b then nat else bool.
Definition Sch2 (b:bool) : Set := P b.
```

Il n'y a évidemment pas de telle dépendance possible en Ocaml. Une solution pour rester uniforme est de fabriquer malgré tout une variable de type qui en fait ne sert jamais :

```
type 'b p = 𝕋
type 'b sch2 = 𝕋 p
```

Nous verrons plus tard qu'il est en fait possible de repérer de telles dépendances vis-à-vis des termes et de les supprimer complètement, afin d'obtenir :

```
type p = 𝕋
type sch2 = p
```

Enfin, considérons maintenant le dernier cas, celui d'une dépendance vis-à-vis d'un nouveau schéma de type :

```
Definition Sch3 : (bool → Set) → Set :=
 fun (X:bool→Set) ⇒ X true → X false.
```

A priori, une variable de type Ocaml est insuffisante pour représenter le schéma de type `X`. Nous devons donc approximer, ce qui peut se faire de plusieurs manières.

- La réponse la plus prudente est de considérer `(X true)` et `(X false)` comme inconnus. En effet, si l'on instancie `X` par la constante `P` définie ci-dessus, on obtient par exemple :

```
Sch3 P = nat → bool
```

  Ceci nous conduit à une première possibilité d'extraction : toute variable qui n'est pas un type est vue comme inconnue, avec ses arguments. Ici :

```
type 'x sch3 = 𝕋 → 𝕋
```

- Mais cette réponse est en pratique trop floue pour être intéressante. En effet, dans une situation non réellement dépendante comme `(Sch3 (fun _ ⇒ nat))`, on obtient au final $\mathbb{T} \to \mathbb{T}$ au lieu de `nat → nat`, or ceci se produit fréquemment en pratique. Par contre, on obtient bien `nat → nat` si l'on prend pour extraction :

```
type 'x sch3 = 'x → 'x
```

  Et concernant les situations dépendantes comme `(Sch3 P)`, cette nouvelle extraction se comporte correctement. Ici par exemple, `P` va être considéré comme un type inconnu $\mathbb{T}$. Et on obtient donc toujours $\mathbb{T} \to \mathbb{T}$.

Entre la première version, plus systématique, et la deuxième, plus fine, nous avons choisi la deuxième.

Les types proprement dits

La construction de type qui est la plus immédiatement traduisible en Ocaml est la flèche fonctionnelle, c'est à dire le produit non dépendant. Ainsi : `nat → nat` est directement exprimable aussi bien en Coq qu'en Ocaml.

Par contre le produit dépendant, lui, n'est en général pas exprimable en Ocaml. En effet il induit une liaison locale d'une variable à l'intérieur d'un type. Regardons par exemple le type de notre `distr_pair` de la section 3.1.

```
((X:Set) X → X) → nat*bool
```

Si l'on essaie à tout prix de générer une variable de type pour `X`, alors cette variable va avoir pour portée l'ensemble du type. En plus de fausser la sémantique du type Coq, cela a des conséquences néfastes : chaque déclaration contenant des produits va voir son nombre de paramètres exploser. La seule traduction raisonnable est en fait d'ignorer la dépendance du produit interne, et de remplacer chaque occurrence de la variable par $\mathbb{T}$. On obtient ici :

```
(𝕋 → 𝕋 → 𝕋) → nat*bool
```

Seuls les produits en tête du type peuvent être traduits à la rigueur (cf. section 3.5).

Pour étudier les autres cas, il faut noter qu'on s'autorise à réduire dans les types lors de leur extraction, afin de limiter le nombre de situations non gérables avec les types Ocaml. Ceci est nouveau par rapport à l'extraction des termes, pendant laquelle toute réduction est évidemment hors de question. Dans la suite du chapitre, tout se fait donc modulo $\beta\iota\zeta$, afin d'être aussi précis que possible. La situation de la δ-réduction, plus complexe, va être évoquée en même temps que celle de l'extraction des constantes de types. Toutes ces réductions sont évidemment dangereuses au niveau de l'efficacité de la fonction d'extraction, mais heureusement en pratique les temps d'extraction restent raisonnables, même sur des exemples conséquents.

On va donc considérer tout type sous sa forme normale de tête, qui a l'allure suivante : $\forall \overrightarrow{x_i : X_i}, (t\ \overrightarrow{a_j})$. On a déjà vu comment traiter les produits en tête. L'extraction du reste se fait selon la structure de la tête t, qui peut être :

- une sorte s (qui est alors sans argument)
- une constante c
- un inductif I
- une variable X
- un `match` non réductible
- un `fix` non réductible

Si cette tête t est une sorte, tout terme de type t est un schéma de type, et son extraction donne $\Box$. Afin de rester cohérent avec cette extraction des termes, le type t doit être extrait vers le type $\Box$.

Considérons maintenant le cas d'une constante c appliquée. Une solution de facilité consiste alors à δ-réduire cette constante. Mais ceci n'est pas satisfaisant. car cela mène en moyenne à des types plus gros et moins lisibles. Et d'autre part, ceci n'est pas faisable dans tous les cas, puisque toutes les constantes n'ont pas forcément un corps utilisable. Il suffit de considérer, par exemple, des constantes abstraites d'une signature de module, ou encore des axiomes[6] On distingue alors trois situations pour les constantes :

1. Le cas le plus favorable est celui où cette constante est un schéma de type. On a alors envie, comme pour `(Sch1 nat)`, de traduire les arguments et de retourner l'application de types Ocaml. A ceci près que les arguments n'ont pas de raison d'être des types :
   - Si un argument est un terme, l'approximation par $\mathbb{T}$ s'impose.
   - Si maintenant un argument `a` est un schéma de type attendant n arguments, on peut tenter de le voir comme un vrai type regroupant tous les devenirs possibles de `a`. Pour cela on l'applique n fois à $\mathbb{T}$, sauf qu'en pratique, au lien de laisser ces

[6]En pratique l'extraction actuelle permet la réalisation des axiomes informatifs par du code fourni manuellement. Mais ce code fourni n'est pas analysé par l'extraction (cf. section 4.4.2).

arguments $\mathbb{T}$, on réduit : cela consiste à supprimer les n lambdas de tête[7] de `a`, et à remplacer par $\mathbb{T}$ les variables créées par ces lambdas.

2. Si maintenant on est dans la situation inhabituelle où un type comporte en tête une constante qui n'est pas un schéma de type, il y a peu de chance de traduire cette constante en une constante de type. La meilleure approche est alors de réduire la constante, si possible, puis d'extraire la version réduite. L'exemple habituel quoique peu réaliste pour illustrer cette situation est celui de l'identité.

```
Definition id := fun (X:Type)(x:X) ⇒ x.
```

Il est ici clair que traduire `(id Set nat)` en $\mathbb{T}$ sous prétexte que la constante de tête n'est pas un schéma de type est trop grossier, alors qu'une étape de δ-réduction mène à `nat`.

3. Enfin pour les constantes non schémas de type et non réductibles, le dernier recours est l'approximation par $\mathbb{T}$.

Le cas d'un type inductif appliqué est similaire à celui d'une constante appliquée, en plus simple étant donné qu'un inductif I a par construction un type de la forme $\forall a : A, \ldots \forall z : Z, s$. On laisse donc toujours l'inductif en tête, et il ne reste plus qu'à extraire les arguments comme précédemment.

En ce qui concerne une variable, tout a déjà été dit précédemment, selon l'origine de cette variable. Si cette variable provient d'un produit dépendant, la variable et ses arguments deviennent $\mathbb{T}$ puisque la dépendance disparaît en Ocaml. Si maintenant cette variable provient des paramètres d'un schéma de type ou d'un inductif, elle donne alors une variable de type Ocaml, ses arguments étant ignorés.

Quant aux derniers cas non encore évoqués pour un type, à savoir ceux des types sous forme de `match` ou `fix` non réductibles, ils sont trop complexes pour Ocaml, et sont donc traduits en $\mathbb{T}$.

3.3.5 La fonction $\widehat{\mathcal{E}}$ d'extraction des types

Nous allons maintenant reprendre de manière formelle les différentes situations décrites jusqu'ici.

Définition 16 *La fonction $\widehat{\mathcal{E}}$ d'extraction des types de* Cci *vers les types Ocaml est définie de façon mutuellement récursive. Elle utilise un ensemble v de variables de types à traduire, noté en indice et initialement vide.*

Commençons par les types proprement dits, c'est à dire les termes Coq acceptant une sorte pour type. Le premier cas concerne les parties logiques.

(prop) si U est de type `Prop`*, alors $\widehat{\mathcal{E}}_v(U) = \Box$.*

Le reste se fait par cas sur la tête du type après $\beta\iota\zeta$-réduction :

(sorte) $\widehat{\mathcal{E}}_v(s) = \Box$

[7] S'il n'y a pas n lambdas en tête de `a`, on en rajoute via η-expansion

(prod1) *si* $\widehat{\mathcal{E}}_v(B) = \square$, *alors* $\widehat{\mathcal{E}}_v(\forall x : A,\ B) = \square$

(prod2) *si* $\widehat{\mathcal{E}}_v(B) \neq \square$, *alors* $\widehat{\mathcal{E}}_v(\forall x : A,\ B) = \widehat{\mathcal{E}}_v(A) \to \widehat{\mathcal{E}}_v(B)$

(case) $\widehat{\mathcal{E}}_v(\texttt{case}(\ldots, \ldots, \ldots)\ \overrightarrow{a_i}) = \mathbb{T}$

(fix) $\widehat{\mathcal{E}}_v((\texttt{fix}\ \ldots)\ \overrightarrow{a_i}) = \mathbb{T}$

(var1) *si* $X \in v$, *alors* $\widehat{\mathcal{E}}_v(X\ \overrightarrow{a_i}) = {}'X$

(var2) *si* $X \notin v$, *alors* $\widehat{\mathcal{E}}_v(X\ \overrightarrow{a_i}) = \mathbb{T}$

(ind1) *si* I *est un type inductif, on pose*[8] *:* $\widehat{\mathcal{E}}_v(I\ a_1\ \ldots\ a_n) = (\widehat{\mathcal{E}}_v(a_1), \ldots, \widehat{\mathcal{E}}_v(a_n))\ I$

(cst1) *si la constante* c *est un schéma de types, alors :*
$\widehat{\mathcal{E}}_v(c\ a_1\ \ldots\ a_n) = (\widehat{\mathcal{E}}_v(a_1), \ldots, \widehat{\mathcal{E}}_v(a_n))\ c$

(cst2) *Dans le cas contraire, et si la constante peut se réduire, on le fait :*
si $(c\ \overrightarrow{a_i}) \to_\delta U$ *alors* $\widehat{\mathcal{E}}_v(c\ \overrightarrow{a_i}) = \widehat{\mathcal{E}}_v(U)$

(cst3) *Enfin dans le cas d'une constante non réductible et non schéma de types, on utilise en dernier ressort une approximation :* $\widehat{\mathcal{E}}_v(c\ \overrightarrow{a_i}) = \mathbb{T}$

Les appels récursifs sur les arguments d'un inductif ou d'une constante de type ne portent pas nécessairement sur des types Coq. *On s'y ramène de la façon suivante :*

(sch) *Si* U *est un schéma de type, on peut donc l'écrire modulo* η*-expansion sous la forme :* $\lambda a : A, \ldots \lambda z : Z, V$, *avec* V *étant un type. On pose alors :* $\widehat{\mathcal{E}}_v(U) = \widehat{\mathcal{E}}_v(V)$, *en ignorant donc les variables.*

(terme) *Si* U *n'est pas un schéma de types, alors* $\widehat{\mathcal{E}}_v(U) = \mathbb{T}$.

Enfin, l'extraction d'un environnement de déclarations Coq *se fait ainsi :*

(nil) $\widehat{\mathcal{E}}([]) = []$

(def1) *Pour la déclaration d'un schéma de type* c *dont le corps peut s'écrire* $t =_\eta \lambda a : A, \ldots \lambda z : Z, U$, *on pose* $v = \{a, \ldots, z\}$ *et*
$\widehat{\mathcal{E}}(\Gamma; (c := t : T)) = \widehat{\mathcal{E}}(\Gamma); (\texttt{type}\ ({}'a, \ldots, {}'z)\ c = \widehat{\mathcal{E}}_v(U))$

(def2) *Pour la déclaration d'une constante* c *non schéma de type, on ne produit rien :* $\widehat{\mathcal{E}}(\Gamma; (c := t : T)) = \widehat{\mathcal{E}}(\Gamma)$

(ax1) *Pour un axiome dont le type* T *est de la forme* $\forall a : A, \ldots \forall z : Z$, s *on produit une déclaration de type abstrait :*
$\widehat{\mathcal{E}}(\Gamma; (ax : T)) = \widehat{\mathcal{E}}(\Gamma); (\texttt{type}\ ({}'a, \ldots, {}'z)\ ax)$

(ax2) *Sinon, pour un axiome dont le type n'est pas de la forme* $\forall a : A, \ldots \forall z : Z$, s *on ne produit rien :* $\widehat{\mathcal{E}}(\Gamma; (ax : T)) = \widehat{\mathcal{E}}(\Gamma)$

(ind2) *Pour un inductif* I *ayant* n *paramètres* $\overrightarrow{p_i}$ *et dont les constructeurs sont* $\overrightarrow{C_i}$, *on pose :*

$$\widehat{\mathcal{E}}(\Gamma; \texttt{Ind}_n((I : \forall \overrightarrow{p_i : P_i},\ \forall \overrightarrow{x_i : X_i},\ s)\ :=\ (\overrightarrow{C_i : T_i})))$$
$$= \widehat{\mathcal{E}}(\Gamma); (\texttt{type}\ (\overrightarrow{{}'p_i}, \overrightarrow{{}'x_i})\ I\ \texttt{=}\ C_1\ \texttt{of}\ \pi(\widehat{\mathcal{E}}_v(T_1))\ |\ \ldots\ |\ C_n\ \texttt{of}\ \pi(\widehat{\mathcal{E}}_v(T_n)))$$

avec $v = \{p_1, \ldots, p_n\}$
$\pi(\tau_1 \to \ldots \to \tau_n \to \tau) = \tau_1 * \ldots * \tau_n$

[8] on utilise en sortie la syntaxe postfixe Ocaml pour les applications de type

Cet algorithme utilisant les réductions Coq termine grâce à la normalisation forte du Cci. Il faut également noter qu'il fournit des types Ocaml bien formés. En particulier chaque application d'un inductif ou d'une constante de type se fait au bon nombre d'arguments. Prenons par exemple le cas d'une constante de type $c : \forall \overrightarrow{x : X},\ s$. Lorsqu'on extrait un type Coq ayant c en tête, on sait que c a exactement n arguments, puisque c'est la seule façon pour faire que $(c\ \overrightarrow{a_i})$ ait pour type une sorte. Et chaque argument a_i va donner un argument extrait $\widehat{\mathcal{E}}(a_i)$ en Ocaml. D'autre part, le nombre de variables de type dans la déclaration de c en Ocaml est bien n. Le cas des inductifs est similaire.

3.4 Extraction, typage et garantie de correction

Comme expliqué précédemment, le typage ainsi assuré par l'action de notre version de l'algorithme $\mathcal{M}$ ne permet pas d'assurer en soi la correction de l'exécution du terme, à cause de la présence des `Obj.magic`. Par contre un terme extrait « brut » $\mathcal{E}(t)$ ne va être modifié par cette passe de typage que via l'insertion de `Obj.magic`. Or ces `Obj.magic` n'ont aucune influence sur l'exécution d'un terme : du point de vue du λ-calcul non typé, ce ne sont que des fonctions identité. Les résultats de correction du chapitre précédent s'appliquent donc également au terme extrait bien-typé final.

3.5 Différences avec l'extraction des types implantée

Pour des raisons de simplicité et de concision, la définition $\widehat{\mathcal{E}}$ est incomplète. Les inductifs mutuels, par exemple, bien que non évoqués, se traitent sans plus de problèmes. Quant aux co-inductifs, la section 4.2 leur est dédiée. Un certain nombre de cas particuliers sont également distingués dans l'implantation, comme les inductifs vides ou singletons et les records. Ces cas seront évoqués dans la section 4.3 sur les optimisations.

3.5.1 Filtrage des paramètres de types

Par rapport à la description de $\widehat{\mathcal{E}}$, il est possible de raffiner la gestion des arguments de types. En effet on a vu que les arguments qui ne sont pas des schémas de types vont être traduits par $\mathbb{T}$. On peut en fait supprimer complètement de tels arguments, car on sait les identifier dans tous les cas, et uniquement grâce au typage. Cela correspond aux règles (ind1) et (cst1). Évidemment, afin de rester cohérent, on doit alors également filtrer les variables de types que l'on génère selon qu'elles correspondent ou non à des schémas de types. Il faut donc alors revoir les règles (def1), (ax1) et (ind2). C'est cette version optimisée qui a été implantée. L'exemple précédent des listes de taille `n` s'extrait donc en fait vers :

```
type 'a listn = Niln | Consn of nat * 'a * 'a listn
```

Ceci correspond aux listes standards, mis à part cet argument `nat` témoin de la taille de la liste, qui doit être conservé.

3.5.2 Gestion des arguments non paramétriques des inductifs

Ensuite, il faut aussi noter que pour des raisons de simplicité, la fonction $\widehat{\mathcal{E}}$ ne gère correctement que les paramètres des inductifs. Reprenons par exemple l'inductif `list2` défini précédemment :

```
Inductive list2 : ∀A:Set, Type :=
 | nil2 : ∀A:Set, list2 A
 | cons2 : ∀A:Set, A → list2 (A*A) → list2 A.
```

Si l'on suit scrupuleusement $\widehat{\mathcal{E}}$, l'extraction de `list2` n'est pas celle souhaitée, mais plutôt :

```
type 'a list2 = Nil2 of 𝕋 | Cons2 of 𝕋 * 𝕋 * (𝕋 * 𝕋) list2
```

En effet $\widehat{\mathcal{E}}$ ne fait pas de lien entre le produit $\forall A$: `Set`, ... dans le type de l'inductif et les produits $\forall A$: `Set`, ... dans les types des constructeurs. Le premier donne un `'a` alors que les autres donnent $\mathbb{T}$. Par contre l'extraction des types actuellement implantée essaie autant que possible de repérer ces « pseudo-paramètres », et sur cet exemple `list2` produira bien :

```
type 'a list2 = Nil2 | Cons2 of 'a * ('a * 'a) list2
```

3.5.3 Réduction de certaines constantes de types

Une autre différence entre $\widehat{\mathcal{E}}$ décrit précédemment et ce qui a été implanté concerne les constantes de type. Pour l'instant, la règle (cst1) traduit toujours une constante schéma de type appliquée en la même constante appliquée. On a vu dans l'exemple `Sch3` précédent que `(Sch3 P)` donnait en Ocaml le type `p sch3`, qui est en fait $\mathbb{T} \to \mathbb{T}$ une fois réduit. Mais si on δ-réduit `(Sch3 P)` au niveau de Coq, on obtient `nat → bool`, qui s'extrait vers lui-même, et on obtient donc un extraction plus précise qu'auparavant. La stratégie actuellement implantée est d'essayer les deux. Considérons un type Coq t qui se δ-réduit en tête vers u :

- si le δ-réduit de $\widehat{\mathcal{E}}(t)$ en Ocaml est égal à $\widehat{\mathcal{E}}(u)$, on garde alors la version la plus compacte, à savoir $\widehat{\mathcal{E}}(t)$.
- sinon on garde la version la plus précise, même si elle n'est pas la plus compacte, à savoir $\widehat{\mathcal{E}}(u)$.

3.5.4 Traitement particulier des produits de tête

Enfin, une dernière optimisation d'implantation concerne les produits de tête. En effet, le rejet de toutes les variables des produits est parfois trop extrême. Ainsi le type de l'identité polymorphe `id` n'est pas celui attendu :

```
Ê(∀X:Set, X → X) = 𝕋 → 𝕋 → 𝕋
```

au lieu de $\mathbb{T}\to$ `'x → 'x`. Ceci est en partie corrigé dans l'implantation :

- Dans le cas d'une extraction de type $\widehat{\mathcal{E}}(T)$, afin de déterminer le type d'un terme extrait $t : T$, on autorise les produits en tête de T à générer des variables de types. Ceci permet d'obtenir des types plus fins pour les termes extraits En particulier, le type de `id` est bien celui espéré.
- L'autre situation est celle d'une extraction de type $\widehat{\mathcal{E}}(T)$ effectuée afin de transformer une déclaration de type Coq en une déclaration de type Ocaml. Dans ce cas, on ne génère aucune variable, pour respecter le principe suivant des schémas de types : autant de variables de types que de lambdas en tête du schéma.

Chapitre 4

L'extraction en pratique : raffinements et implantation

Dans ce chapitre, nous allons décrire l'état actuel de l'extraction telle qu'elle est implantée dans la version 8.0 de Coq. Cette description va se décomposer en deux parties.

Tout d'abord, nous allons présenter un certain nombre d'avancées implantées dans l'outil d'extraction, mais non encore évoquées jusqu'ici. Ces avancées seront présentées uniquement de façon informelle, car leur étude théorique n'est pour l'instant pas aussi complète que celle de la fonction $\mathcal{E}$ du chapitre 2. Ces raffinements sont :

- le support du nouveau système de modules Coq,
- le support des co-inductifs,
- l'ajout de plusieurs optimisations destinées à améliorer l'efficacité du code extrait.

Dans un deuxième temps, nous allons présenter l'implantation qui a été réalisée au cours de cette thèse, décrire succinctement son fonctionnement interne, et surtout son maniement du point de vue de l'utilisateur.

4.1 Extraction des nouveaux modules Coq

Au cours de cette thèse, une nouveauté importante est apparue dans Coq, à laquelle l'extraction a pu être adaptée. Il s'agit du nouveau système de modules, qui apporte une souplesse inconnue jusqu'ici en terme de structuration des développements, de raisonnement abstrait et de de réutilisation de code et/ou de preuves. Du point de vue de l'extraction, ce système de modules ouvre de nouvelles perspectives en terme de preuves de programmes, en facilitant la conception d'éléments certifiés autonomes. Qu'il s'agisse de modules ou de foncteurs, ces parties certifiées peuvent alors aisément s'assembler via leurs interfaces avec d'autres parties, certifiées ou non, afin de constituer une application de plus grande envergure. L'extraction des modules va en fait prolonger l'effort décrit au chapitre précédent visant à obtenir une interface `.mli` pour tout fichier extrait `.ml`. Désormais, toutes les structures extraites, modules et foncteurs, auront leurs interfaces.

A ce titre, le développement que nous présentons au chapitre 7 constitue une première pierre vers l'édification d'une bibliothèque de modules certifiés, utilisables par tout programmeur, qu'il désire bâtir une application également certifiée, ou tout simplement réutiliser des

briques de base de qualité. Dans ce développement concernant les structures d'ensembles finis, nous certifions plusieurs implantations de foncteurs ayant la même interface, ce qui permet à toute application bâtie sur cette interface de faire son choix parmi différentes implantations, dont plusieurs certifiées. D'autre part, nous avons établi de nombreuses propriétés dérivées à partir de l'interface Coq des ensembles finis. Ces propriétés sont donc automatiquement partagées par toutes les implantations de cette interface, ce qui permettra, espérons-le, de certifier aisément des programmes utilisant cette bibliothèque d'ensembles finis.

Avant de passer maintenant à une présentation de ces nouveaux modules Coq et de leur extraction, nous allons commencer tout d'abord par un rapide survol du système de modules d'Ocaml, qui a inspiré son pendant Coq.

4.1.1 Les modules Ocaml

Le système de modules d'Ocaml [52] est lui-même dérivé du système original de SML [42]. Celui-ci a également évolué en retour, et les deux systèmes sont maintenant très voisins et connus sous le nom de système de modules Harper-Lillibridge-Leroy. On trouvera une présentation complète de ce système dans le manuel de référence d'Ocaml [53]. De façon sommaire, on peut dire que ce système de modules ajoute trois sortes de structures par-dessus les déclarations de base du langage que sont les déclarations de types et les déclarations de constantes :

- Une première structure ajoutée est celle de *module*. Il s'agit d'un groupement de déclarations qui peuvent correspondre à une valeur, un type ou éventuellement une nouvelle sous-structure modulaire.

```
module UnModule = struct
 type 'a unType = UnConstructeur of 'a
 let uneFonction = function UnConstructeur x → x
end
```

  Le point important est que les déclarations internes au module partagent un même espace de noms, distinct de celui des noms du niveau externe. On accède alors aux objets internes via la notation qualifiée, par exemple ici `UnModule.unType`.

- Les *signatures* ou interfaces forment une deuxième sorte de structures. Elles sont également constituées d'un groupement de déclarations, mais ces déclarations ne concernent que le typage. Ces signatures permettent ainsi de typer les modules précédents. Par exemple, si l'on soumet à Ocaml la déclaration du module `UnModule`, celui-ci nous renvoie une signature correspondante :

```
# module UnModule = struct
   type 'a unType = UnConstructeur of 'a
   let uneFonction = function UnConstructeur x → x
                              .../...
```

```
.../...
  end;;
module UnModule :
 sig
   type 'a unType = UnConstructeur of 'a
   val uneFonction : 'a unType → 'a
 end
```

Autant le corps des modules est délimité par les mots-clés `struct...end`, autant une signature est introduite par `sig...end`. Dans le cas particulier de la signature précédente, inférée par Ocaml, on y retrouve chaque déclaration de types à l'identique, ainsi qu'une construction `val` pour chaque déclaration de valeurs du module d'origine. Cette signature est donc la plus précise possible. Mais d'autres signatures sont également admissibles pour typer notre module :

- Par rapport à la signature la plus précise, on peut enlever certaines déclarations. On obtient alors des objets locaux au module, inaccessibles depuis l'extérieur.
- On peut également cacher le contenu de certains types, pour en faire des types *abstraits*, dont les objets seront uniquement manipulables via les primitives fournies dans le module. Ici :

```
module type UneSignatureRestreinte = sig
 type 'a unType
 val uneFonction : 'a unType → 'a
end
```

- Enfin, les *foncteurs* sont une généralisation des modules. Un foncteur est une sorte de fonction prenant des modules en arguments et fabriquant un nouveau module :

```
module UnFoncteur = functor (M:UneSignatureRestreinte) → struct
 type 'a unAutreType = ('a M.unType) list
 let uneAutreFonction l = List.map M.uneFonction l
end
```

Le corps de ce foncteur est donc paramétré par rapport à une variable de module `M`. Et tout module admettant la signature `UneSignatureRestreinte` pour type peut alors être appliqué à `UnFoncteur`, prendre la place de cette variable de module `M` et donner un nouveau module :

```
# module UnNouveauModule = UnFoncteur(UnModule);;
module UnNouveauModule :
 sig
   type 'a unAutreType = 'a UnModule.unType list
   val uneAutreFonction : 'a UnModule.unType list → 'a list
 end
```

Pour conclure cette brève présentation des modules Ocaml, il faut évidemment mentionner que ces modules Ocaml présentent bien d'autres subtilités que nous ne décrirons pas

ici, comme par exemple le mécanisme des `with` dans les types de modules. Enfin, pour des exemples d'utilisation des modules plus subtils que ceux de cette section, on pourra consulter le chapitre 7 sur une formalisation des ensembles finis par le biais de modules.

4.1.2 Les modules Coq

Cela fait longtemps que l'on souhaitait enrichir Coq d'un système de modules digne de ce nom. Il existait bien précédemment des outils destinés à organiser un développement de manière modulaire, comme par exemple le découpage possible en plusieurs fichiers, le mécanisme de `Section` à l'intérieur d'un fichier ou encore l'utilisation d'axiomes. Mais ces méthodes atteignaient vite leurs limites. Ainsi, à partir d'un développement paramétré par un axiome, la seule façon de réutiliser ce développement en le spécialisant à un cas concret était de dupliquer son code et de remplacer l'axiome par la définition concrète. On retrouve ce procédé par exemple dans le projet C-CoRN (voir chapitre 6) pour sa structure des nombres réels.

La première étude en vue d'un système de modules pour Coq a été faite par J. Courant dans sa thèse [22]. Mais son système, plus ambitieux que celui d'Ocaml, a semblé être trop complexe pour être implanté par-dessus le code existant de Coq. A la place, un système à la Ocaml a finalement été implanté par J. Chrząszcz [18, 19], en s'inspirant du système modulaire de modules de X. Leroy [52].

On retrouve donc désormais en Coq des modules, des signatures et des foncteurs. Par exemple, un équivalent de notre module jouet précédent peut s'écrire ainsi en Coq :

```
Module UnModule.
 Inductive unType (A:Set) : Set := UnConstructeur : A → unType A.
 Definition uneFonction (A:Set)(a:unType A) : A :=
  match a with UnConstructeur x ⇒ x end.
End UnModule.
```

On peut voir qu'à la différence de Ocaml, les modules Coq sont *interactifs*. Alors qu'en Ocaml, un module doit être fourni en une déclaration d'un seul tenant, Coq permet de le construire étape par étape, par des déclarations successives entre la commande de début (`Module UnModule`) et celle de fin (`End UnModule`). Ce besoin d'interactivité découle de la présence possible de théorèmes, et donc de preuves, parmi les déclarations présentes dans un module. Sauf pour les chanceux dont la langue maternelle est le λ-calcul, il est alors illusoire d'espérer donner une preuve non-triviale directement. Cette différence, bien qu'importante du point de vue de l'utilisateur, n'influe pas sur l'extraction, qui se déroule toujours après la fin des déclarations de modules.

Les signatures et les foncteurs se définissent de la même manière :

```
Module Type UneSignature.
 Parameter unType : Set → Set.
 Parameter uneFonction : ∀A:Set,(unType A) → A.
End UneSignature.

Module UnFoncteur (M:UneSignature).
 Definition unAutreType := fun A ⇒ list (M.unType A).
 Definition uneAutreFonction := fun A l ⇒ List.map (M.uneFonction A) l.
End UnFoncteur.
```

Au niveau du contenu d'une signature, on s'éloigne quelque peu du parallèle avec les signatures Ocaml, à cause des différences entre les deux systèmes, et en particulier l'absence de distinction entre termes et types en Coq. Rappelons qu'en Ocaml un terme est obligatoirement abstrait via la déclaration `val`, alors qu'une déclaration `type` peut être abstraite ou non.

- En ce qui concerne les déclarations abstraites dans une interface Coq, le mot-clé unifié[1] est `Parameter`, qu'il s'agisse de notre fonction `uneFonction` ou de son type `unType`.
- Écrire un type concret dans une interface Coq est sans surprise : on retrouve ainsi la déclaration d'un inductif via `Inductive`, ou bien la définition d'un alias via `Definition`.
- La principale nouveauté concerne la possibilité de placer un terme concret dans une interface. Ainsi, `Definition x:=0` dans une interface obligera toute implantation de cette interface à contenir une constante `x` valant `0`.

Deux détails pratiques rendent délicat l'usage des interfaces Coq. Tout d'abord, seul un type `unType` défini comme une constante (via `Definition` par exemple) est accepté comme réalisation du paramètre `unType`, et *pas* un type défini par `Inductive`. Ainsi, pour Coq, `UneSignature` n'est *pas* une signature valide de `UnModule`. La solution est alors de faire un alias au niveau du module `UnModule` : on renomme `unType` en `unType'` au niveau de la définition inductive, puis on ajoute la déclaration suivante :

```
Definition unType := unType'.
```

D'autre part, il est à noter que si l'on impose une signature contenant des déclarations abstraites `Parameter` à notre module (par exemple via `Module UnModule : UneSignature`), alors le corps de notre fonction `uneFonction` est caché : on ne peut plus jamais s'en servir hors du module, par exemple lors d'un calcul. Cela diffère de la sémantique d'Ocaml : les valeurs des constantes sont effectivement cachées dans les interfaces par la construction `val`, mais on se sert évidemment de ces valeurs lors de l'exécution du programme ! La relation Coq de typage « `:` » entre modules et interface étant donc très contraignante, on lui préfère souvent une forme plus faible « `<:` », qui se contente de vérifier que l'interface *pourrait* être une signature valide du module, mais sans mettre en place les restrictions associées.

[1] En fait, `Axiom` convient également, c'est même ici un synonyme.

Tout comme lors de la présentation des modules Ocaml, il reste beaucoup à dire concernant les possibilités avancées des modules Coq. Et encore une fois, le chapitre 7 présente des exemples bien plus réalistes d'usage de ces modules et foncteurs.

4.1.3 L'extraction des modules

Voyons maintenant ce que devient l'extraction en présence de ces modules Coq. Tout d'abord, signalons que seule l'extraction vers Ocaml accepte de traiter les modules Coq. En effet, il n'y a pas en Haskell d'équivalent direct d'un tel système de modules, mais plutôt un système de *classes*, notion assez différente. Peut-être est-il possible d'utiliser ces classes pour exprimer les modules Coq extraits, mais la faisabilité de cette traduction n'a pas été explorée.

Considérons maintenant l'extraction des modules Coq vers Ocaml. En première approximation, tout se passe simplement : une signature Coq devient une signature Ocaml, un module Coq devient un module Ocaml et de même pour les foncteurs.

Une fois ces évidences affirmées, le point délicat est la traduction des déclarations internes à ces structures modulaires. En effet on souhaite préserver lors de l'extraction les relations de typage entre modules et signatures. Pour traduire une déclaration Coq en une déclaration Ocaml correspondante, nous allons utiliser les fonctions d'extraction $\mathcal{E}$ et $\widehat{\mathcal{E}}$ des chapitres 2 et 3, qui produisent respectivement des termes et des types.

Nous allons maintenant détailler les différentes situations possibles. En fait, il n'y a que quatre types de déclarations possibles en Coq :

- la déclaration d'un inductif via `Inductive`.
- la déclaration d'une constante, par exemple via `Definition`. Toutes les déclarations `Lemma`, `Theorem`, `Fixpoint` ne sont en effet que des variantes de `Definition` permettant une saisie plus aisée du corps de la constante que l'on définit.
- la spécification d'un objet dans une signature via `Parameter`.
- la déclaration d'un axiome dans un module, via `Axiom`. Nous ignorerons ce cas pour l'instant, voir la section 4.4.2 pour le traitement des axiomes par l'extraction.

Les deux premiers cas peuvent survenir aussi bien sous un module (ou foncteur) que sous une signature.

Déjà, toutes les déclarations d'objets de sorte `Prop`, qu'il s'agisse de types inductifs ou de constantes, vont être purement et simplement ignorées par l'extraction. En effet aucune déclaration extraite ultérieure ne peut dépendre d'une telle déclaration logique. S'il s'agit par exemple de la déclaration d'une constante logique `c` de sorte `Prop`, toute occurrence ultérieure de `c` sera placée par la fonction $\mathcal{E}$ sous une constante $\Box$. Et si `c` apparaît dans un terme en position de type et est donc traité par la fonction $\widehat{\mathcal{E}}$, alors toute référence à `c` disparaît également, car il ne s'agit pas d'un type. Quant à un type inductif logique `I`, la fonction $\mathcal{E}$ le fait disparaître comme tous les types, et la fonction $\widehat{\mathcal{E}}$ le transforme en un type dégénéré $\Box$.

Ensuite, un cas simple est celui d'une déclaration de types inductifs informatifs. En effet, nous allons alors tout simplement reprendre la déclaration associée générée par la fonction $\widehat{\mathcal{E}}$. Et ceci va s'appliquer que l'on soit dans une signature ou sous un module.

Considérons maintenant une déclaration informative `Parameter x:T` dans une signature, qui réclame donc l'existence d'un objet `x` de type `T` dans tout module implantant cette signature. Ocaml, à la différence de Coq, distingue types et termes. Cette déclaration peut donc correspondre en Ocaml soit à la déclaration d'une valeur `val x:`$\widehat{\mathcal{E}}$`(T)`, soit à la déclaration d'un type abstrait `type x`. Et ce choix se fait selon l'allure de `T` : si `T` est un type comme `nat`, on s'oriente vers la déclaration d'une constante `val x:nat`. Si au contraire `T` est une sorte, ou plus généralement une arité, `x` est un type (resp. un schéma de types), et sa traduction naturelle est une déclaration de type en Ocaml.

Enfin, la dernière situation est celle de la déclaration d'une constante, typiquement via `Parameter x:T:=t`. On réutilise alors la distinction entre les schémas de types et les autres termes. Pour les premiers, on génère une déclaration de type en utilisant la fonction $\widehat{\mathcal{E}}$, ce qui donne `type x =` $\widehat{\mathcal{E}}$`(t)`. Pour les seconds, on engendre une déclaration de terme Ocaml. Et cette déclaration prend alors soit une forme concrète si l'on est dans un module ou un foncteur, à savoir `let x =` $\mathcal{E}(t)$, soit une forme abstraite `val x :` $\widehat{\mathcal{E}}$`(T)` si l'on est dans une signature.

Si l'on récapitule toutes ces situations possibles, on obtient le tableau suivant :

	sorte `Prop`	schéma de types	cas standard
`Inductive i := C:T`	$\varnothing$	`//////////////`	`type i = C of` $\widehat{\mathcal{E}}$`(T)`
`Definition x:T:=t`	$\varnothing$	`type x =` $\widehat{\mathcal{E}}$`(t)`	Module : `let x =` $\mathcal{E}$`(t)` Signature : `val x :` $\widehat{\mathcal{E}}$`(T)`
`Parameter x:T`	$\varnothing$	`type x`	`val x :` $\widehat{\mathcal{E}}$`(T)`

Même si ce tableau récapitulatif ne le montre pas afin de rester simple, toute déclaration de type Ocaml, qu'elle provienne d'un inductif ou d'un schéma de type, est susceptible de contenir des variables de type `'a`, comme nous l'avons vu dans le chapitre 3.

Une fois décrit cet « aiguillage » des objets Coq vers soit les termes, soit les types Ocaml, on doit alors vérifier que cet aiguillage est bien cohérent. Ainsi, pour tout objet Coq nommé `c` qui se retrouve placé au niveau des termes par l'extraction, toutes les occurrences de `c` dans les objets extraits ultérieurement doivent être également au niveau des termes. Et de même pour un objet placé au niveau des types. Cette propriété n'est pas si immédiate. Prenons par exemple l'identité polymorphe :

```
Definition id := fun (X:Type)(x:X) ⇒ x.
```

Comme il ne s'agit pas d'un schéma de types, notre extraction choisit d'en faire une fonction Ocaml :

```
let id _ x = x
```

Mais `id` peut tout à fait servir dans un type Coq :

```
Definition nat_bis := id Set nat.
```

Une possibilité d'extraction de ce type `nat_bis` serait alors l'application `nat id`, avec `id` faisant ici référence à une constante de type `id` qui pourrait être définie par :

```
type 'x id = 'x
```

Or notre extraction ne définit pas cette constante de type, mais seulement la constante `id` au niveau des termes. En fait ici tout fonctionne bien, grâce aux règles que nous avons choisies pour l'extraction $\widehat{\mathcal{E}}$ des types en section 3.3.5. En effet, l'extraction $\widehat{\mathcal{E}}$`(nat_bis)` n'est pas `nat id`, mais `nat`, car `id` n'est pas un schéma de type. On réduit donc cette constante `id` avant d'extraire, selon la règle (cst2) définissant $\widehat{\mathcal{E}}$.

Plus généralement, notre « aiguillage » est cohérent avec les deux fonctions $\mathcal{E}$ et $\widehat{\mathcal{E}}$ car ces trois opérations partagent le même critère de stratification, à savoir le fait d'être ou non un schéma de types. Ainsi une constante de type `c` ne peut apparaître dans un terme extrait car en tant que schéma de type, elle se retrouvera sous un $\Box$. Et à l'opposé, une constante de terme comme `id` ne peut subsister dans un type extrait d'après les règles (cst1) (cst2) et (cst3) de $\widehat{\mathcal{E}}$.

4.1.4 Limitations actuelles de l'extraction des modules

Cette extraction des modules Coq est encore à considérer comme expérimentale. Tout d'abord, une étude théorique poussée n'a pas encore été réalisée faute de temps. Ceci supposerait en particulier l'intégration à notre système théorique des chapitres 1 et 2 les règles de typage propres aux modules Coq. Or ces règles occupent actuellement à elles seules 6 pages du Manuel de Référence [78] (chapitre 5).

L'implantation, qui est donc en avance sur la théorie concernant ces modules, donne en pratique des résultats satisfaisants sur les premiers développements réels utilisant ces modules. On pourra par exemple se reporter au chapitre 7 pour une étude de cas autour des ensembles finis. Mais nous avons dans le même temps identifié deux situations limites pouvant mettre le code actuel en défaut.

Un premier problème de typage

Comme nous l'avons mentionné précédemment, une propriété essentielle de l'extraction des modules est la conservation des relations de typages entre un module et une signature lors de l'extraction. Ceci est par exemple critique pour qu'une application de foncteur continue à être possible après extraction. A première vue, cette propriété semble être une simple conséquence du résultat suivant reliant les fonctions $\mathcal{E}$ et $\widehat{\mathcal{E}}$:

$$t : T \Rightarrow \mathcal{E}(t) : \widehat{\mathcal{E}}(T)$$

Malheureusement, ce résultat ne parle que de types concrets, et la présence de types abstraits dans une signature peut venir perturber la situation. En fait, il est actuellement possible de construire un module `Mod` et une signature `Sig` contredisant notre propriété de conservation du typage des modules. Pour cela, il suffit de combiner des types abstraits et des schémas de types donc l'extraction va être approchée :

```
Module Type Sig.
 Parameter t : nat → Set.
 Parameter x : t 0.
End Sig.

Module Mod : Sig.
 Definition t := F.
 Definition x := 0.
End Mod.
```

Le schéma de type `F` est celui de la page 91 : `(F n)` est le type des fonctions entières d'arité `n`, et donc `(F 0)` est le type des fonctions entières à 0 arguments, c'est-à-dire `nat`. Or l'extraction de `Mod` et `Sig` donne :

```
module type Sig =
 sig
  type t
  val x : t
 end

module Mod =
 struct
  type t = 𝕋
  let x = 0
 end
```

En effet, le schéma de type `F` défini par point-fixe donne le type extrait le plus général $\mathbb{T}$, et au niveau abstrait `t 0` est approximé en `t` puisqu'on ne peut réduire le type abstrait `t`. On voit au final qu'après extraction, `Sig` n'est plus une signature valide pour `Mod`.

Pour corriger ce problème, on peut imaginer que $\mathbb{T}$ devienne réellement un type universel pour Ocaml, c'est à dire que l'on ait `t`:$\mathbb{T}$ pour tout terme Ocaml `t`, et en particulier `0`:$\mathbb{T}$. Mais ceci supposerait une modification du système de type d'Ocaml. Sinon une solution est ici d'insérer un `Obj.magic` autour du `0` dans `Mod`. Le problème est que cette insertion d'`Obj.magic` ne se fait plus selon une analyse locale (le type de `x` à l'intérieur du module), mais selon une analyse globale, par exemple de savoir si `Mod` est utilisé plus tard avec la signature `Sig` dans une application de foncteur. On pourrait aussi utiliser une encapsulation au moment de l'application de foncteur, comme :

```
module Mod_bis =
 include Mod
 let x = Obj.magic x
end
```

Aucune de ces solutions n'a été mise en place pour l'instant. De toute façon, il faut relativiser l'importance de ce problème. Il s'agit certes d'un accroc à l'objectif des « 100% des constructions Coq extractibles de façon bien typée ». Mais jusqu'à maintenant, aucun déve-

loppement réaliste n'est tombé dans ce cas de figure précis, à savoir la combinaison de types abstraits et de schémas de types dans une signature. Et même si le faible nombre actuel de tels développements modulaires rend une extrapolation difficile, il est vraiment très peu probable que le problème arrive un jour en pratique.

Un second problème de typage

En fait il est également possible d'induire des problèmes de typage entre modules et signature en utilisant la cumulativité :

```
Module Type Sig2.
 Parameter t:Type.
 Parameter x:t.
End.

Module Mod2 : Sig2.
 Definition t:Prop:=True.
 Definition x:True:=I.
End.
```

La signature se traduit naturellement en :

```
module type Sig2 =
 sig
  type t
  val x : t
 end
```

Par contre le module extrait `Mod2` est vide, car `Mod2.t` et `Mod2.x` sont respectivement un type et une valeur logique. Et nous avons fait le choix précédemment de supprimer de telles déclarations qui ne servent jamais dans des déclarations ultérieures. Or ici, le typage de `Mod2` par `Sig2` exige la présence de champs `t` et `x` dans `Mod2`. Mais ce problème de typage est moins gênant que le premier. Après tout, rien n'empêche de revenir sur notre choix d'éliminer complètement les déclarations logiques, ce qui donnerait ici :

```
module Mod2 =
 struct
  type t = 𝕋
  let x = 𝕋
 end
```

Mais l'extraction laisserait alors quantité de déclarations logiques parasites ne servant jamais sauf en cas d'usage de la cumulativité entre un module et sa signature. Il vaut mieux alors continuer à expurger modules et signatures des déclarations logiques et remédier aux situations hautement exceptionnelles utilisant la cumulativité via une encapsulation semblable à celle résolvant le problème précédent. D'ailleurs, la correction de ces deux situations à problèmes sera sans doute à faire de manière conjointe.

En fait, nos soucis de typage des modules présentent une grande similarité avec les problèmes de typage au niveau des termes. De même que nous avons besoin de fonctions non-typées de coercion `Obj.magic`, il faudrait disposer de fonctions de coercion au niveau des modules, lors par exemple de l'application d'un module à un foncteur. Mais à la différence du niveau des termes, de telles fonctions n'existent pas au niveau des modules.

Terminons tout de même cette discussion sur le typage des modules extraits par deux remarques positives :

- L'extraction actuelle vérifie au moins une forme faible de conservation du typage des modules : si un module `M` admet `S` comme signature la plus générale (celle inférée par le système), alors l'extraction de `M` admet encore l'extraction de `S` comme signature la plus générale. Et plus généralement, en l'absence de types abstraits et de cumulativité entre module et signature, alors tout se passe bien.
- Pragmatiquement, si le vérificateur de types d'Ocaml est satisfait par le résultat d'une extraction de structures modulaires, alors tout est pour le mieux. En particulier les résultats précédents de correction pour l'exécution ou la sémantique d'un terme extrait sont toujours valides. Après tout, les modules et les foncteurs ne sont qu'une manière de réutiliser du code. Et comme les applications des foncteurs sont connues statiquement, on peut obtenir du code équivalent sans foncteurs grâce à un procédé nommé défonctorisation, qui a par exemple été implanté pour Ocaml par J. Signoles [77].

Un problème de nommage

Il existe un autre problème, syntaxique cette fois, pouvant mener à des modules extraits refusés par Ocaml. En effet Coq, avec ses modules interactifs, est plus tolérant qu'Ocaml quant aux possibilités de nommage des objets. En particulier on peut faire référence au module même que l'on est en train de construire, ce qui est illégal en Ocaml. L'exemple suivant est légal en Coq :

```
Module M.
 Definition t := 0.
 Module N.
  Definition t := 1.
  Definition u := M.t
 End N.
End M.
```

Par contre en Ocaml on ne peut utiliser le nom qualifié `M.t` à l'intérieur de `M`, et le nom simple `t` est incorrect à cause de la présence dans l'espace de nom local du `t` correspondant à `N.t`. Le renommage pouvant être ardu à cause des éventuelles signatures à respecter, une solution raisonnable, signalée par J. Signoles, est alors d'ajouter un module local :

```
module M =
 struct
                                    .../...
```

```
...
  let t = O
  module AdHoc = struct let t = t end
  module N =
   struct
    let t = S O
    let u = AdHoc.t
   end
 end
```

Détecter le besoin de tels modules et les ajouter proprement dans tous les cas possibles (termes et types) étant très lourd, ceci n'est pas encore implanté à l'heure actuelle. Mais comme pour les problèmes précédents, il est a priori très peu probable de tomber dans une telle situation dans un développement réaliste.

4.2 Types co-inductifs et extraction

4.2.1 Des inductifs aux co-inductifs

Les types inductifs que nous avons manipulé jusqu'ici ne peuvent contenir que des objets *finis* au sens où ils ne peuvent contenir qu'un nombre fini de constructeurs de ce type[2]. Cette finitude ou bonne fondation implique alors l'existence de principes de récurrence associés à chacun de ces types inductifs. Ces principes de récurrence sont en fait engendrés automatiquement par Coq, comme par exemple `nat_rec` pour `nat`. Maintenant, à la suite des travaux de E. Giménez, il existe également en Coq des types similaires aux inductifs, mais pour lesquels il n'y a pas de contrainte de finitude. Il s'agit des types co-inductifs (voir par exemple [37]). L'exemple typique d'un tel type est celui des flots :

```
CoInductive Stream (A:Set) : Set := Cons : A → Stream A → Stream A.
```

Lorsqu'on définit un type co-inductif, la première différence avec la déclaration d'un type inductif est que Coq ne peut plus engendrer de principe de récurrence associé, ces principes n'étant plus valides pour des objets infinis.

Sur un objet co-inductif l'analyse par cas fonctionne comme pour un objet inductif. Voici par exemple comment accéder à la tête d'un flot :

```
Definition hd (A:Set) (x:Stream A) :=
 match x with
  | Cons a _ ⇒ a
...
```

[2]On peut noter au passage qu'un type inductif Ocaml comme `list` ne vérifie *pas* la même propriété de finitude que le type `list` de Coq. En effet, Ocaml autorise des objets cycliques infinis comme `let rec l = 0::l`. Et évidemment, si l'on applique cette liste à une fonction extraite comme `map`, le calcul bouclera. Ceci n'invalide en rien les résultats théoriques du chapitre 2, puisque `l` ne peut être mis en correspondance via $\widehat{\texttt{list}}$ avec aucune liste `l'` de Coq.

```
end.
```

Par contre la récursion est bien différente dans le cas co-inductif. Tout d'abord on parle plutôt de co-récursion, et le mot-clé Coq est `CoFixpoint` au lieu de `Fixpoint`[3]. Ensuite il n'y a pas de notion d'argument de décroissance comme pour le `Fixpoint`. On peut ainsi écrire :

```
CoFixpoint from (n:nat) : Stream nat := Cons n (from (S n)).
```

Ou encore, sans aucun argument :

```
CoFixpoint zero_stream : Stream nat := Cons 0 zero_stream.
```

Mais toute définition n'est pas autorisée pour autant, car il est pour le moins délicat de donner un sens à une définition telle que :

```
CoFixpoint dummy : Stream nat := dummy.
```

La règle est d'autoriser uniquement les co-récursions qui bâtissent effectivement un nouvel objet co-inductif. Plus précisément tout appel co-récursif doit être situé sous au moins un constructeur co-inductif, ce qui n'est pas le cas pour notre `dummy`.

Quant à la réduction d'un co-point-fixe, elle suit également une règle particulière, afin de ne pas briser la propriété de normalisation forte du système. Il s'agit d'une réduction *paresseuse* : un co-point-fixe ne peut être déplié et remplacé par son corps que si ce co-point-fixe et ses éventuels arguments apparaissent en position de tête d'un filtrage. Par exemple `(from 0)` est en forme normale, tandis que dans `(hd (from 0))`, comme `hd` est un filtrage, on peut déplier `from` une fois, ce qui donne après simplification `0` comme forme normale.

4.2.2 L'extraction des types co-inductifs

Le cas Haskell

Une fois n'est pas coutume, les types co-inductifs forment une caractéristique de Coq qui a d'abord été prise en compte par l'extraction vers Haskell avant d'être intégrée à l'extraction Ocaml. La raison est bien sûr le caractère paresseux de l'évaluation d'Haskell, qui rend triviale l'extraction des co-inductifs vers ce langage. Ainsi, il n'y a pas de différence entre l'extraction des listes finies et celle des flots hormis la présence ou non du cas de base `Nil`.

```
data List a = Nil
            | Cons a (List a)

data Stream a = Cons a (Stream a)
```

Ensuite l'extraction d'un co-point-fixe donne naturellement une fonction récursive :

```
from n =
```

[3]Il existe aussi un `cofix` anonyme correspondant au `fix`.

.../...

```
 Cons n (from (S n))

zero_stream =
 Cons O zero_stream
```

Tant que `zero_stream` ne sera pas nécessaire à un calcul ultérieur, cette constante ne sera jamais dépliée. Et même alors, il n'y aura jamais de dépliage superflu. Ainsi, si l'on demande l'affichage de `(hd (tl (tl zero_stream)))`, il n'y aura que trois dépliages, menant à un résultat de `O`.

Historiquement, l'extraction des co-inductifs vers Haskell était déjà opérationnelle dans l'ancienne extraction. Nous n'avons fait que maintenir cette possibilité. A titre d'exemple, on pourra se reporter à la contribution utilisateur nommée `Rocq/MUTUAL-EXCLUSION`. E. Giménez y étudie l'exclusion mutuelle de deux processus via l'algorithme de Petersson. Une petite interface graphique utilisant la bibliothèque Fudgets de Haskell permet de visualiser le déroulement de l'algorithme de façon interactive.

Le cas Ocaml

Au cours de la réalisation de notre nouvelle extraction, nous avons ajouté la possibilité d'extraire des co-inductifs vers Ocaml. Ceci se fait via un encodage, car une traduction directe et naïve de nos exemples précédents serait incorrecte, compte tenu de l'évaluation strict de Ocaml. Si l'on prend :

```
type 'a stream = Cons of 'a * 'a stream

let rec from n = Cons n (from (S n))
```

Alors le calcul de `(hd (from O))` part dans une boucle infinie d'appels à `from`.

Heureusement, il existe en Ocaml un mécanisme pour introduire des objets paresseux. Ainsi `(lazy x)` est une version interrompue du calcul de l'objet `x`. Et si ce `x` a pour type `a`, alors `(lazy x)` a pour type `a Lazy.t`. Enfin la fonction `Lazy.force : 'a Lazy.t → 'a` permet de forcer la reprise du calcul.

Nous allons donc nous servir de ce mécanisme pour encoder les types co-inductifs de Coq. Chaque type `t` va, en fait, être extrait vers deux types `t` et `__t`, définis mutuellement, le premier étant le type des objets en attente, et le second le type des objets débloqués. Ainsi pour les flots, cela nous donne :

```
type 'a stream = 'a __stream Lazy.t
and 'a __stream = Cons of 'a * 'a stream
```

Et on simule maintenant la réduction Coq de la façon suivante. Un constructeur co-inductif produit un objet en attente, et est donc entouré du mot-clé `lazy`. A l'opposé, un filtrage sur un objet co-inductif nécessite d'accéder à la structure superficielle de cet objet. on force donc un niveau de calcul en insérant un `Lazy.force` autour de l'objet filtré.

Ceci nous donne l'extraction Ocaml suivante pour nos exemples :

```
let hd x = match Lazy.force x with
```

.../...

```
...
  | Cons (a,s) ⇒ a

let rec from n =
  lazy (Cons (n,(from (S n))))

let rec zero_stream =
  lazy (Cons (O,zero_stream))
```

Cet encodage est inspiré du style « even, with difficulty » présenté dans [81]. Le nommage « even » fait référence au fait que tout constructeur co-inductif est associé à un `lazy`, doublant ainsi le nombre de constructions syntaxiques. Et le « with difficulty » signale que ce style s'oppose à un autre, plus simple mais pouvant mener à des évaluations superflues d'éléments dans un flot. Nous allons voir maintenant que le style choisi pour l'extraction peut aussi poser des problèmes d'évaluation superflue et donc d'efficacité.

Co-inductifs, Ocaml et efficacité

L'utilisation des constructions `lazy` et `Lazy.force` par l'extraction ne fait que retarder l'évaluation des constructions co-inductives. Mais elle ne transforme aucunement Ocaml en un langage complètement paresseux. En particulier, les côtés stricts d'Ocaml peuvent resurgir, en particulier au niveau de l'évaluation des arguments de fonction, et mener à des différences significatives avec l'évaluation en Haskell de nos exemples à base de co-inductifs.

Considérons par exemple une fonction `iter` qui, à partir d'une fonction `f` et d'un point initial `a`, calcule le flot constitué de `a`, `(f a)`, `(f² a)`, etc.

```
CoFixpoint iter (A:Set)(f:A→A)(a:A): Stream A := Cons a (iter A f (f a)).
```

Son extraction est alors :

```
let rec iter f a = lazy (Cons (a,(iter f (f a))))
```

Mais alors demander l'évaluation de `(hd (iter f a))` conduit à l'évaluation superflue de `(f a)`, même si l'appel récursif `(iter f (f a))` est bien bloqué par le `lazy` en tête de `iter`. Cette évaluation superflue, peu naturelle, peut être fâcheuse si `f` conduit à des calculs longs.

En même temps, il ne s'agit pas à proprement parler d'un problème de correction de l'extraction. En effet, si l'on considère la réduction en Ocaml de `(hd (iter f a))`, on peut simuler cette réduction par une réduction similaire au niveau Coq, de la même façon que lors des résultats théoriques du chapitre 2. Cela signifie que Coq pourrait très bien choisir de normaliser également `(f a)` lors de la réduction de `(hd (iter f a))`. En pratique Coq ne le fait pas, ayant une stratégie d'évaluation par défaut plutôt paresseuse.

Nous entrons donc, avec cet exemple, dans le domaine des questions d'efficacité du code extrait, domaine que nous détaillerons dans la section suivante, et qui bien souvent n'a que des solutions imparfaites.

Ici, dans l'exemple qui nous occupe, on peut tout à fait déplacer le `lazy` afin de bloquer l'évaluation des arguments de l'appel récursif :

```
let rec iter f a = Cons (a, lazy (iter f (f a)))
```

La fonction `iter` ne fabrique plus alors une `stream` mais une `__stream`. Mais outre ces petits ajustements de types qu'elle implique, cette solution correspond au style « odd » de l'article [81]. Et ce style souffre également de problèmes d'évaluation superflue. On peut également tenter de bloquer à deux endroits l'évaluation de `iter` :

```
let rec iter f a = lazy (Cons (a, lazy (Lazy.force (iter f (f a)))))
```

De façon plus générale, il semble être intéressant d'insérer des points de blocage supplémentaires `lazy (Lazy.force (...))` dans le code extrait autour des sous-expressions de types co-inductifs. Ceci n'est pas fait automatiquement, mais rajouter manuellement de tels points de blocages peut être fait sans risque, car cela ne modifie pas la correction du code.

Une situation voisine de celle du `iter` précédent a été rencontrée lors de la réalisation d'une extraction Ocaml de la contribution `Rocq/MUTUAL-EXCLUSION`. Un point de blocage est alors ajouté via un usage détourné de la commande `Extract Constant`, qui sera présentée à la fin de ce chapitre. Le lecteur intéressé par plus de détails pourra consulter les fichiers de cette contribution.

Enfin, signalons qu'il faudrait également insérer `lazy (Lazy.force (...))` autour du corps des co-points-fixes sans arguments ne commençant pas par un constructeur. En effet, autant Ocaml accepte notre point-fixe sans argument `zero_stream`, autant il refusera des corps plus complexes, commençant par exemple par un `if`. Cacher ces corps trop complexes pour Ocaml sous un `lazy` permet de contourner la difficulté.

4.3 Extraction et optimisations de code

Nous allons maintenant décrire un certain nombre de transformations que l'extraction effectue sur le code extrait afin de tenter d'en améliorer l'efficacité, ou parfois simplement la lisibilité.

4.3.1 Suppression de certains arguments logiques

La première de ces transformations est destinée à faire disparaître autant que possible les résidus logiques laissés par la fonction $\mathcal{E}$ d'extraction du chapitre 2. Ces résidus[4] prennent la forme d'abstractions anonymes `fun _ → ...` et de constantes □. Nous avons déjà vu qu'en Haskell, □ peut être réalisé par une erreur (voir page 82). Par contre en Ocaml la possibilité d'avoir à réduire □ `x` vers □ nous force à utiliser une définition complexe (voir page 103) :

```
let __ = let rec f _ = Obj.repr f in Obj.repr f
```

Il est alors évidemment désirable de se passer autant que possible de telles constantes □, aussi bien du point de vue de l'efficacité que de la lisibilité du code extrait.

[4] En fait, ces résidus peuvent également provenir de schémas de types et pas seulement de parties logiques, mais nous allons amalgamer ces deux cas en parlant simplement de « résidus logiques ».

Prenons l'exemple d'une division `div` avec pré-condition, mais sans post-condition, dont le type Coq est `∀a b:nat, b≠0 → nat`. L'extraction telle que nous l'avons vue jusqu'ici produit alors un fonction `div` de type `nat → nat → □ → nat` . Et toute utilisation ultérieure de cette fonction pour effectuer une division aura la forme `(div a b □)`.

Maintenant, si l'on doit conserver de tels résidus logiques, c'est que l'évaluation en Ocaml des applications partielles pourrait sinon mener à des situations anormales (voir 2.1). L'argument logique résiduel de `div` permet ici de faire que `(div a b)` soit une clôture bloquée en attente de son troisième argument.

Mais la grande majorité des applications rencontrées par l'extraction sont en fait totales. Il est donc préférable de faire qu'une application totale de fonction soit aussi simple et naturelle que possible, quitte à alourdir l'écriture des applications partielles.

Tentons alors de rendre à notre fonction extraite `div` son type naturel `nat → nat → nat`. Au niveau de la définition de `div`, ceci ne pose pas de problème. Modulo η-expansion, on peut en effet supposer que cette définition de `div` commence par `fun a b _ →...`. Il suffit alors d'enlever la troisième abstraction. Une fois simplifiée ainsi la définition de `div`, il faut évidemment adapter également les appels ultérieurs à cette fonction :

- Considérons tout d'abord une application totale `(div a b □)`. Pour s'adapter à notre nouvelle fonction `div` épurée, il suffit de jeter le troisième argument □. Dans ce cas, on rejoint le code produit par l'ancienne extraction : seuls les arguments informatifs subsistent, les arguments logiques disparaissant complètement. Et ceci ne se fait pas au détriment de la sécurité : si l'on déplie `div` dans cette application avant et après transformation, on voit qu'on est passé de :

```
(fun a b _ → ...) a b □
```

  à la nouvelle forme :

```
(fun a b → ...) a b
```

  Ces deux formes sont clairement équivalentes du point de vue de l'évaluation.

- Prenons maintenant dans Coq une application partielle `(div a₀ b₀)`, et donc une extraction produisant à l'origine une application partielle de la forme `(div a b)`. Lors du passage à notre nouvelle version de `div`, nous devons impérativement maintenir le caractère « bloqué » de cette application partielle. Pour cela nous l'adaptons en `(fun _ → (div a b))`. Encore une fois, déplier `div` montre que l'on n'a pas modifié la sémantique de l'application :

```
(fun a b _ → ...) a b
```

  est devenu

```
(fun _ → ((fun a b → ...) a b))
```

  La seule conséquence est de retarder l'évaluation de `a` et `b`. On pourrait imaginer une adaptation encore plus fidèle, à savoir :

```
let x = a and y = b in fun _ → (div x y)
```

Mais en pratique passer à cette version n'a pas semblé être nécessaire.

- Pour le cas d'applications encore plus partielles, comme `(div a)` et enfin `div` sans aucun argument, on procède de même : `(div a)` devient `(fun b _ → (div a))` et `div` seul devient `(fun a b _ → (div a b))`.

Bien sûr, ce que nous avons décrit ici pour `div` se généralise en la transformation de n'importe quelle déclaration de fonction ayant un nombre arbitraire d'arguments, et des arguments logiques à des positions également arbitraires. Ainsi la déclaration d'une fonction `f` de type $\mathtt{t}_1 \to \ldots \to \mathtt{t}_n \to \mathtt{t}$ va être transformée en la déclaration d'une fonction de type $\mathtt{t}_{i_1} \to \ldots \to \mathtt{t}_{i_p} \to \mathtt{t}$ où les $\mathtt{t}_{i_k}$ sont tous les $\mathtt{t}_i$ différents de $\Box$.

La seule exception à cette démarche concerne les fonctions n'ayant que des arguments logiques, comme par exemple `False_rec` (voir 2.1). Supprimer tous les arguments et faire de l'extraction de `False_rec` une constante ne serait pas correct, puisqu'ici `False_rec` correspond à une situation anormale, et donc à la levée d'une exception via `assert false` en Ocaml. La déclaration suivante

```
let false_rec = assert false
```

interromprait alors le programme final dès son lancement. Ce problème se corrige en gardant toujours au moins un argument à nos fonctions, ce qui nous donne ici :

```
let false_rec _ = assert false
```

La transformation présentée ici ne permet évidemment pas à elle seule de supprimer tout résidu logique. Par exemple un filtrage ayant certaines de ses branches informatives et certaines logiques continuera à produire de $\Box$ lors de l'extraction. D'autre part, il s'agit uniquement d'une simplification de premier niveau : nous supprimons ou non certains arguments des fonctions traitées, mais nous ne modifions en rien le type de ces arguments. Ainsi une fonction de type $\mathtt{nat} \to \Box \to \mathtt{nat}$ aura comme nouveau type $\mathtt{nat} \to \mathtt{nat}$, mais une fonction de type $(\mathtt{nat} \to \Box \to \mathtt{nat}) \to \mathtt{nat}$ restera inchangée.

Étendre cette transformation pour permettre d'expurger l'intérieur des types a été envisagé un moment, puis abandonné : cela impliquerait en effet des manipulations de termes devenant de plus en plus complexes à chaque niveau supplémentaire traité.

En pratique, la transformation limitée actuellement implantée permet déjà d'éliminer une très grande partie des résidus logiques. On se retrouve donc avec une extraction qui est le plus fréquemment compatible avec l'ancienne extraction, et ce même sur des exemples pouvant être complexes, tout en gardant sa propriété de correction, même sur les exemples les plus pathologiques.

Signalons enfin qu'en Haskell cette élimination pourrait être bien plus poussée, car dans ce langage, la présence de résidus logiques à des fins de blocage de réduction n'a pas lieu d'être. Pour l'instant, ceci n'a pas été fait, l'extraction Haskell et l'extraction Ocaml partageant actuellement une large base commune pour des raisons de simplicité.

4.3.2 Optimisations des types inductifs

Nous allons tout d'abord voir comment l'élimination des arguments logiques présentée précédemment pour les fonctions s'applique également aux constructeurs des types inductifs. Ensuite, nous verrons deux cas particuliers de types inductifs qui font l'objet d'un traitement particulier par l'extraction, à savoir les inductifs singletons et les records (ou enregistrements).

Constructeurs et élimination des arguments logiques

De même que les arguments de type □ des fonctions, les arguments de type □ des constructeurs inductifs vont également être supprimés, car ils sont superflus. Prenons par exemple l'inductif informatif `sig` présenté initialement page 28. Pour mémoire, le type `(sig A P)` exprime l'existence d'un objet `x` de type `A` vérifiant le prédicat `P`, ce qu'on note aussi avec la syntaxe `{x:A|P x}`. Et l'unique constructeur de `sig` admet quatre arguments : `(exist A P x p)` est de type `(sig A P)` lorsque `x` est le témoin recherché et `p` est une preuve de `(P x)`. La version brute de l'extraction de `sig` est alors[5] :

```
type 'a sig0 = Exist of □ * □ * 'a * □
```

En fait, les deux premiers arguments `A` et `P` de `exist` sont des `paramètres` de l'inductif `sig`. Les règles de typage du Cci assure alors que ces paramètres ne peuvent varier dans la définition des constructeurs de `sig`. Ces paramètres n'apportent donc en fait aucun nouveau contenu calculatoire, et sont systématiquement supprimés des définitions inductives par l'extraction. Ici cela ne change rien, car ces paramètres sont des types, qui auraient donc été enlevés par le mécanisme dont nous allons parler maintenant. Mais même un paramètre informatif, par exemple de type `nat`, sera supprimé par l'extraction.

Il reste alors le cas du quatrième argument, qui est un terme logique. Mais il est en fait immédiat de débarrasser la définition extraite de ce □ inutile :

```
type 'a sig0 = Exist of 'a
```

On doit alors adapter les utilisations de ce type :

- Au niveau d'une application au constructeur `Exist`, on peut supposer la présence des quatre arguments. En effet la syntaxe décurryfiée de Ocaml l'exige, ce qui fait que l'extraction fabrique au besoin les arguments manquants par η-expansion. Il ne reste plus alors qu'à filtrer les arguments, ici en ne gardant que le troisième.
- Au niveau d'un filtrage comme `match e with Exists(a,p,x,q)` $\rightarrow$ `t`, les propriétés de l'extraction assurent que `a`, `p` et `q`, qui ont tous le type □, ne vont pas apparaître dans `t`. On remplace donc ce filtrage par `match e with Exists(x)` $\rightarrow$ `t`.

Et ce qui a été présenté ici pour `sig` se généralise à tout inductif, quel que soit son nombre de constructeurs, de paramètres ou d'arguments. On peut remarquer qu'il n'y a pas besoin ici d'un traitement particulier dans le cas où il ne resterait aucun argument à un constructeur après transformation.

[5] `sig` est renommé en `sig0` par l'extraction puisque `sig` est un mot-clé Ocaml.

Simplification des inductifs singletons informatifs

Dans le cas particulier de l'extraction de l'inductif `sig`, on peut en fait pousser plus loin la simplification. En effet on constate qu'il ne reste qu'un seul argument à `Exist` après la transformation précédente. Comme `sig` n'a qu'un seul argument, sa version extraite n'est plus maintenant qu'une simple encapsulation. Il est alors préférable de supprimer cette couche d'encapsulation. On va alors simplement convertir le type `(a sig0)` en `a`, et le terme `Exist(t)` en `t`. Enfin il reste à traduire le filtrage sur un objet de type `sig` : le terme `match e with Exist(x) → t` correspond alors à `let x = e in t`.

Cette simplification s'étend aux types dits singletons informatifs, c'est-à-dire n'ayant qu'un seul constructeur, et dont cet unique constructeur n'a plus qu'un seul argument après extraction[6].

Ce traitement des inductifs singletons informatifs permet de gagner en occupation mémoire, en temps de calcul et également en lisibilité. En outre, on obtient alors le comportement attendu de l'extraction vis-à-vis de la quantification existentielle informative : à partir d'une preuve Coq dont l'énoncé est de la forme `∀x:t, P x → ∃y:u, Q y`, on obtient bien alors une fonction de type `t → u`, et non plus de type `t → u sig0`.

Les records

Une autre catégorie de types inductifs reçoit un traitement particulier de la part de l'extraction. Il s'agit des types inductifs définis via la déclaration `Record` de Coq. Par exemple :

```
Record paire (A B:Set) : Set := { gauche : A ; droite : B }.
```

Au niveau interne, ces records Coq ne sont pas primitifs, mais traduits vers des types inductifs. Notre exemple est ainsi enregistré par Coq sous la forme :

```
Inductive paire (A B:Set) : Set := Build_paire : A → B → paire A B.
```

L'avantage de la déclaration `Record` est que Coq engendre automatiquement[7] à partir de cette déclaration deux fonctions projections `gauche : (paire A B) → B` et `droite : (paire A B) → B`.

Lors de l'extraction, il est tout à fait possible d'ignorer que le type `paire` a été défini comme un record. Ce type serait alors extrait comme un type inductif standard :

```
let ('a,'b) paire = Build_paire of 'a *'b
```

Et les projections associées serait alors des filtrages, par exemple :

```
let gauche = function
```

.../...

[6]Pour être tout à fait précis, il ne faut pas non plus que le type de cet argument unique fasse intervenir l'inductif de départ, sinon la simplification est erronée.

[7]Il faut noter que la génération de certaines fonctions de projection est parfois impossible pour des raisons de typage.

.../...

```
  | Build_paire (x, y) → x
```

Il en fait préférable de tirer parti des possibilités d'Ocaml [8], en utilisant sa syntaxe primitive pour les enregistrements. Ceci permet alors d'accéder directement à un champ d'un enregistrement via la notation « pointée », ici par exemple `p.gauche`, ce qui est légèrement plus efficace qu'une projection par filtrage. En outre, la lisibilité du code extrait en est améliorée. Illustrons le passage aux records primitifs d'Ocaml avec notre petit exemple :

- Le type extrait est maintenant :

```
type ('a,'b) paire = { gauche : 'a; droite: 'b }
```

- Les fonctions de projection ne sont alors fournies qu'au cas où elles seraient utilisées sans argument :

```
let gauche x = x.gauche
```

- Et pour chaque projection munie de son argument, plutôt que de l'écrire sous forme fonctionnelle `(gauche p)`, on produit maintenant `p.gauche`.
- Chaque usage du constructeur `Build_paire`, comme `(Build_paire g d)`, devient un enregistrement `{gauche=g; droite=d}`. Comme lors des transformations précédentes d'inductifs, toute application partielle de ce constructeur aura été d'abord η-expansée.
- Enfin, tout filtrage sur un type devenant un record subit l'adaptation suivante :

```
match p with Build_paire (x,y) → ...
```

devient :

```
match p with {gauche=x; droite=y} → ...
```

À l'origine, le besoin d'une extraction améliorée des records s'est fait sentir lors des premiers essais d'extraction du projet C-CoRN (voir le chapitre 6 pour plus de détails). En effet, ce développement utilise abondamment des structures algébriques définies via des records. Et ces structures sont reliées par des coercions qui sont en fait des projections. Au niveau du développement Coq, ces coercions restent implicites, donc invisibles. Mais au niveau du code extrait, la moindre addition de réels dans cette formalisation fait alors intervenir 7 projections. Tout gain d'efficacité et de lisibilité à ce niveau est alors appréciable.

4.3.3 Dépliage du corps de certaines fonctions

Les transformations présentées jusqu'ici pouvaient modifier les types des objets extraits. Nous allons maintenant passer à une deuxième catégorie de transformations, qui elles conservent le type des objets extraits. La première de ces transformations va consister à remplacer certains noms de fonctions par le corps de ces fonctions. Autrement dit, lors

[8] Il existe également en Haskell des enregistrements primitifs, mais l'extraction ne s'en sert pas encore.

de l'extraction, nous allons anticiper la δ-réduction de ces fonctions. Évidemment, de tels dépliages ne sont pas à effectuer systématiquement, sous peine de rendre le code extrait gigantesque. Mais nous allons voir que dans certains cas ciblés, ces dépliages sont cruciaux pour l'efficacité du code extrait en Ocaml. En fait il existait déjà un tel mécanisme de dépliage dans l'ancienne extraction. Nous nous en sommes inspiré en modifiant ses critères. En outre ce mécanisme antérieur n'a jamais été documenté à notre connaissance.

Étudions tout d'abord le principe de récurrence[9] `bool_rect` associé aux booléens. Cette fonction est engendrée automatiquement par Coq lors de la définition du type `bool`. Elle a pour type :

```
∀P:bool→Type, P true → P false → ∀b:bool,P b
```

Et son extraction est :

```
let bool_rect f f0 = function
  | True → f
  | False → f0
```

Dans cette définition, `f` correspond à la preuve de `(P true)` et `f0` à la preuve de `(P false)`. On constate naturellement que selon la valeur du troisième argument booléen appliqué à cette fonction, seul un des termes `f` et `f0` va réellement servir. Or tout application `(bool_rect a a' b)` conduit en Ocaml à l'évaluation de `a` et `a'`, compte tenu de la stratégie stricte d'évaluation des arguments de ce langage. Bien sûr, cela peut aussi être anodin si `a` et `a'` sont des valeurs simples comme 0, ou bien des clôtures fonctionnelles dont l'évaluation s'arrête immédiatement. Mais si `a` et `a'` mènent à des calculs coûteux, cela peut être fâcheux.

Une solution pour éviter ces évaluations superflues est alors de remplacer `bool_rect` par sa définition, puis d'évaluer symboliquement ses arguments. Ceci va alors avoir pour effet de pousser les arguments `a` et `a'` sous le filtrage, dans des branches qui s'excluent l'une l'autre :

```
bool_rect a a' b
```

devient :

```
(fun f f0 b = match b with True → f | False → f0) a a' b
```

ce que l'on simplifie immédiatement en :

```
match b with True → a | False → a'
```

Cette présence d'arguments réduits par Ocaml même s'ils sont en fait inutiles est systématique dans les principes de récurrence des inductifs à plus de deux constructeurs. Mais on peut également rencontrer cette situation dans bien d'autres cas de figure. La stratégie de l'extraction vers Ocaml est alors la suivante :

- Tous les principes de récurrence sont dépliés, même pour les inductifs à moins de deux constructeurs. En effet, outre l'intérêt en terme d'efficacité, cela conduit expérimentalement à un code extrait plus proche de ce qu'aurait écrit un programmeur humain, et

[9] En fait, il y en a trois, chacun sur une des sortes de Coq. Mais nous ne considérerons que celui sur `Type`, nommé `bool_rect`. Les deux autres `bool_rec` et `bool_ind` ne sont que des alias du premier.

donc plus lisible. En effet, les tactiques `elim` et `induction` utilisent systématiquement les principes de récurrences et non les points-fixes et filtrages sous-jacents.

- Pour une autre fonction, on effectue un dépliage lorsque les deux conditions suivantes sont remplies :
  - le corps de la fonction n'est pas trop gros, *i.e.* en pratique de taille inférieure à une limite arbitrairement fixée à une dizaine de constructions syntaxiques.
  - certains arguments sont détectés comme potentiellement non-utiles, par exemple présents sous une seule branche d'un filtrage.

Bien sûr, il s'agit là d'un compromis heuristique entre des exigences opposées, qui peut certainement être encore affiné. En attendant une investigation plus complète sur ce domaine, nous avons ajouté la possibilité à l'utilisateur de contrôler plus finement ces dépliages :

- Avec la commande `Set` (resp. `Unset`) `Extraction AutoInline`, l'utilisateur peut activer (resp. désactiver) notre mécanisme automatique de dépliage.
- Il est également possible de forcer le dépliage ou le non-dépliage d'un objet `t` particulier avec `Extraction Inline t` ou `Extraction NoInline t`.

Pour illustrer le côté critique du dépliage pour l'efficacité, nous allons étudier maintenant un problème concret apparu un jour où une manipulation malencontreuse dans le code source de Coq avait désactivé le dépliage automatique. Un exemple extrait[10], qui normalement s'exécutait en quelques secondes, s'est mis à ne pas terminer, même au bout d'une dizaine d'heures. La fonction en cause était une fonction comparant deux entiers de Peano, `lt_eq_lt_dec`. A l'époque, cette fonction était bâtie par une double récurrence sur ses deux arguments[11]. Ceci mène lors de l'extraction à un terme contenant deux niveaux de `nat_rect` qui est le principe de récurrence associé au type `nat` :

```
let lt_eq_lt_dec n m =
  nat_rect
    (fun m0 → nat_rect (Inleft Right) (fun m1 iHm → Inleft Left) m0)
    (fun n0 iHn m0 → nat_rect Inright (fun m1 iHm → iHn m1) m0)
    n m
```

Les détails du code de cette fonction importent peu, hormis ce double niveau de `nat_rect`. Voici la même fonction lorsque le dépliage automatique fonctionne :

```
let rec lt_eq_lt_dec n m =
  match n with
    | O → (match m with
             | O → Inleft Right
```

.../...

[10] Il s'agissait de la contribution Rocq/COC.

[11] En fait la deuxième récurrence est ici inutile et a été remplacée depuis par une simple analyse par cas. Ce changement dans la preuve règle d'ailleurs le problème d'efficacité de la version extraite non-dépliée. Mais qui se soucie de tels détails lorsque l'objectif premier est de finir une preuve ?

```
                                                    .../...
              | S n0 → Inleft Left)
    | S n0 →
        (match m with
           | O → Inright
           | S n1 → lt_eq_lt_dec n0 n1)
```

Si l'on regarde cette version dépliée, il est clair que la comparaison entre deux entiers de Peano `n` et `m` va entraîner un nombre d'appels récursifs `min(n,m)` à cette fonction. Par contre la version non-dépliée précédente a un comportement bien différent. Cela peut sembler étrange, car même si `nat_rect` va évaluer ses arguments, ce ne sont ici que des fonctions. Mais si l'on étudie dans le détail la suite des appels de fonctions (avec `#trace` par exemple), on s'aperçoit que ces fonctions évaluées inutilement vont quand même rencontrer des arguments, se réduire, et au final faire exploser la complexité.

La récursivité étant cachée dans la fonction `nat_rect`, nous allons ici compter le nombre d'appels à cette dernière. Voici le tableau récapitulant le nombre de ces appels lors de la comparaison des entiers `n` et `m` :

m \ n	0	1	2	3	4	5	6	7	8	9	10
0	**2**	2	2	2	2	2	2	2	2	2	2
1	2	**3**	4	5	6	7	8	9	10	11	12
2	2	3	**5**	8	12	17	23	30	38	47	57
3	2	3	5	**9**	16	27	43	65	94	131	177
4	2	3	5	9	**17**	32	58	100	164	257	387
5	2	3	5	9	17	**33**	64	121	220	383	639
6	2	3	5	9	17	33	**65**	128	248	467	849
7	2	3	5	9	17	33	65	**129**	256	503	969
8	2	3	5	9	17	33	65	129	**257**	512	1014
9	2	3	5	9	17	33	65	129	257	**513**	1024
10	2	3	5	9	17	33	65	129	257	513	**1025**

On constate en particulier que comparer un entier `n` avec lui-même entraîne $2^n + 1$ appels à `nat_rect`. En fait, si l'on nomme $\mathcal{N}_n^m$ ces nombres d'appels, on constate alors que $\mathcal{N}_n^m - \mathcal{N}_n^{m-1} = C_n^m$. Les colonnes de ce tableau sont donc des sommes partielles de C_n^m qui partent de 2, et l'on retrouve ainsi le cas particulier de la diagonale. Nous ne justifierons pas plus ces formules donnant $\mathcal{N}_n^m$, mais il n'est pas (trop) difficile d'établir par exemple $\mathcal{N}_n^n = 2^n + 1$ à partir du code de `lt_eq_lt_dec`.

En tout cas, comparer de cette façon 30 et 32 engendre plus d'un milliard d'appels de fonctions ! Le dépliage de `nat_rect` transforme ici une fonction exponentielle en une fonction linéaire.

4.3.4 L'amélioration de code : optimisation ou ralentissement ?

La morale de l'exemple précédent est qu'il est très délicat de prévoir quelle sera la forme du code extrait lorsque l'on est face à un script de preuve fait de suites de tactiques comme `induction` ou `auto`. Et bien souvent le code extrait brut ne correspond pas du tout à ce

qu'un humain aurait écrit. Outre le problème de lisibilité, ceci peut entraîner alors des problèmes d'efficacité, comme nous venons de le voir. Et ce problème est particulièrement flagrant avec un langage strict comme Ocaml.

Outre son problème avec `nat_rect`, la fonction précédente `lt_eq_lt_dec` illustre également un cas de code extrait « idiot ». Sans une des optimisations de l'extraction, l'extraction de cette fonction contiendrait à la place de l'appel récursif `(lt_eq_lt_dec n0 n1)` un sous-terme valant :

```
match lt_eq_lt_dec n0 n1 with
 | Inleft x →
   (match x with
     | Left → Inleft Left
     | Right → Inleft Right)
 | Inright → Inright
```

Évidemment, déconstruire un objet pour le reconstruire à l'identique n'a pas d'intérêt, même si cela n'est ici guère coûteux. Et aucun programmeur n'écrira spontanément un tel code. La raison de ces filtrages surprenants est à chercher du côté des parties logiques, effacées lors de l'extraction. En effet les constructeurs `left`, `right` et `inright` des types inductifs `sumbool` et `sumor` prennent au départ chacun un argument logique. Et les filtrages d'origine en Coq servent en fait à modifier ces arguments logiques. L'extraction essaie alors de simplifier de tels filtrages.

Plus généralement, l'extraction applique un certain nombre de transformations pour tenter de redonner un aspect plus habituel au code produit. Voici le détail de ces transformations :

0. (code=1) Élimination interne des résidus logiques.
1. (code=2) Amélioration de l'affichage des fonctions définies par point-fixe.
2. (code=4) Lorsqu'un filtrage se trouve en tête d'un autre filtrage, on permute l'ordre de ces deux filtrages s'il en résulte des simplifications dans chaque nouvelle branche.
3. (code=8) Un filtrage sur `bool`, `sumbool` ou `sumor` reconstruisant ce qu'il vient de déconstruire est supprimé
4. (code=16) Cette optimisation étend la précédente à tous les types de filtrages.
5. (code=32) Un filtrage dont toutes les branches produisent le même résultat est remplacé par ce résultat.
6. (code=64) Si toutes les branches d'un filtrages débute par une abstraction, on déplace cette abstraction devant le filtrage.
7. (code=128) On propage les arguments d'un filtrage à l'intérieur de ses branches.
8. (code=256) On permute les applications et les let-in.
9. (code=512) Une abréviation `let x=u in t` est dépliée si `x` apparaît une fois au plus dans `t`.
10. (code=1024) Un β-redex `((fun x → t) u)` est réduit si `x` apparaît une fois au plus dans `u`.

Si l'on souhaite désactiver toutes ces transformations, il suffit d'utiliser la commande `Unset Extraction Optimize`. Si l'on souhaite à l'inverse revenir à la situation par défaut où (presque) toutes les transformations sont activées, on peut utiliser la commande `Set Extraction Optimize`. Entre ces deux extrêmes, la version 8.0 de Coq permet d'affiner ses préférences via la commande `Set Extraction Flag n`, où `n` est un entier entre 0 et 2047, correspondant à la la somme des codes binaires des transformations que l'on souhaite activer. Par exemple, si l'on souhaite tout activer sauf la transformation n° 4, de code 16, on utilisera $\texttt{n} = 2047 - 16 = 2031$. C'est d'ailleurs cette valeur qui est la situation par défaut à laquelle ramène `Set Extraction Optimize`, et nous allons voir pourquoi dans un instant.

On peut se demander quel est l'intérêt de pouvoir choisir ainsi « à la carte » le comportement de l'extraction. Le problème est que ces transformations ne sont pas toujours des optimisations, étant donné qu'elles reposent sur des heuristiques qui n'ont pas fait l'objet d'études approfondies, et peuvent tout à fait se révéler néfastes dans certains cas. Pire, une transformation s'est récemment révélée périlleuse pour le typage des termes extraits.

Voyons maintenant en détail quels bénéfices et soucis on peut attendre de chaque transformation.

0. Ces éliminations internes de résidus logiques sont a priori toujours bénéfiques.
1. Cet embellissement des points-fixes peut parfois remanier les arguments des appels récursifs internes, rendant ces appels moins efficaces.
2. Cette permutation de deux filtrages peut soit faire croître la taille du code, soit la diminuer, selon les simplifications qu'elle engendre dans chaque branche.

3&4. Il s'agit de la suppression a priori bénigne des filtrages « idiots » évoqués précédemment. Mais prenons l'exemple suivant :

```
type 'a t = T

let f a = match a with T → T
```

Il est alors très tentant de changer `f` en `let f a = a`, mais son type n'est plus alors que `'a t →'a t` et non plus `'a t →'b t`. Un remède rapide à ce problème a été de couper en deux cette transformation, avec d'un côté une partie sûre n'agissant que sur des types connus comme `sumbool`, et de l'autre une partie non-sûre désactivée par défaut. En pratique l'optimisation n° 3 suffit à simplifier les cas usuels de filtrages « idiots ». Et d'autre part l'optimisation n° 4 n'est pas si dangereuse : un utilisateur activant cette option a de grandes chances d'obtenir du code correctement typé, qui est alors tout à fait correct.

5. Cette optimisation est normalement sans inconvénient.
6. Remonter des λ-abstraction devant un filtrage peut entraîner de multiples évaluations de la tête du filtrage, mais peut aussi améliorer la suppression des résidus logiques par les autres transformations.
7. Cette option duplique du code, mais peut aussi aider à propager des simplifications.
8. Un terme comme `((let x = t in u) v)` est peu naturel, et cache le fait que `v` est moralement un argument de `u`, ce qui empêche d'appliquer éventuellement d'autres

simplifications. Il semble alors anodin de permuter l'application et le let-in, afin d'obtenir `let x = t in (u v)`. Malheureusement, nous avons rencontré il y a peu une situation où cette permutation, combinée à des η-expansions destinées à palier des limitations de Ocaml, produisait en fait une permutation entre une fonction et un let-in, ce qui correspond au cas suivant, qui n'est pas toujours souhaitable.

9. Ce dépliage d'un let-in, même linéaire, est discutable, vu qu'un calcul initialement factorisé peut se retrouver dans une fonction, et donc effectué plusieurs fois. Mais ce calcul peut aussi se retrouver dans une branche qui n'est jamais évaluée. D'autre part, les fonctions générées par des tactiques contiennent très souvent des applications partielles comme `let g = f x in g y`. Ceci peut mener au calcul de nombreuses clôtures intermédiaires inutiles, et empêche le compilateur Ocaml d'optimiser l'appel total `f x y` comme il sait le faire.

10. Il s'agit de la même problématique que le cas précédent.

On constate donc que ces transformations sont à manier avec beaucoup de précautions. Toute la difficulté est que le code Coq à extraire peut aussi bien provenir de preuves par tactiques que de fonctions écrites directement dans le langage fonctionnel de Coq. Dans le premier cas, la difficulté de contrôler précisément le λ-terme de preuve sous-jacent fait qu'il faut s'attendre à de nombreuses manipulations avant d'obtenir du code extrait raisonnable. Mais dans le second cas, la meilleure chose à faire est de ne rien modifier afin de respecter les choix de l'utilisateur. De toute façon, il est illusoire d'espérer toujours trouver le code le plus efficace, d'où l'intérêt de permettre à l'utilisateur de faire ses choix parmi les transformations proposées.

L'extraction fait actuellement le choix de privilégier l'amélioration du code provenant des preuves, au détriment éventuellement du code directement écrit. Toutes les optimisations possibles sont donc activées, sauf celle pouvant perturber le typage à savoir la n° 4.

Cette situation actuelle des optimisations n'est que peu satisfaisante au final :

- pas de garantie de correction autre qu'une inspection visuelle des règles de transformation.
- amélioration de l'efficacité et de la lisibilité en moyenne, mais présence de situations particulières néfastes.
- possibilité d'un réglage manuel, mais qui reste assez grossier, et au niveau d'un fichier entier.

Cette partie mériterait donc à elle seule une étude approfondie. Finalement, l'extraction n'est ici qu'un générateur automatique de code. Peut-être faut-il déplacer le problème de l'optimisation au niveau du compilateur ou d'un outil générique de pré-traitement du code source. En même temps, le code extrait possède un certain nombre de traits particuliers qu'un outil générique négligerait peut-être, comme l'absence de parties impératives, ou encore l'importance des applications partielles. En outre, il serait sans doute intéressant de se livrer à une recherche de code-mort dans le code extrait, les parties informatives non élaguées par l'extraction pouvant en contenir.

4.4 État de l'implantation actuelle

4.4.1 Description du code

L'implantation de la nouvelle extraction de Coq réalisée dans le cadre de cette thèse a débuté en 2001 avec l'aide initiale de J.-C. Filliâtre. Cette implantation a été améliorée progressivement, depuis la version 7.0 de Coq jusqu'à la 8.0 actuelle. Elle est actuellement dans un état raisonnable de stabilité et de fonctionnalité, même si par exemple la section précédente montre qu'un certain nombre d'optimisations sont à revoir.

Cette implantation est située dans le sous-répertoire `contrib/extraction` des sources de Coq, et se compose actuellement de 3 700 lignes d'Ocaml. La figure 4.1 présente le graphe de dépendance des fichiers composants cette implantation. Plus précisément :

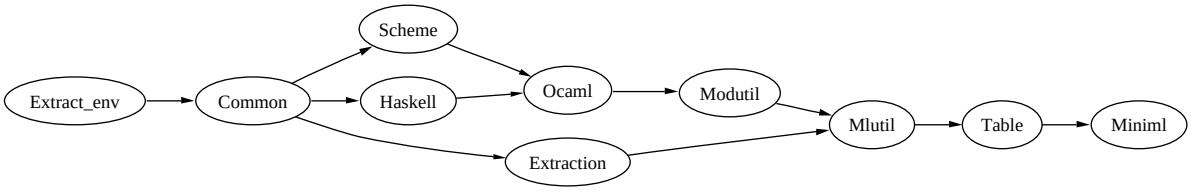


FIG. 4.1: Graphe de dépendance de l'implantation

- L'interface `Miniml` définit les arbres de syntaxe abstraite des termes et types d'un langage MiniML qui va servir de cible commune à l'extraction.
- Le module `Table` est la « mémoire » de l'extraction, il permet par exemple de retrouver l'extraction d'un inductif déjà rencontré.
- Le module `Mlutil` est une boîte à outils de manipulation des termes et types du MiniML. On y définit par exemple la substitution de termes, l'unification de types, ou encore diverses optimisations des termes.
- Quant à `Modutil`, il contient des fonctions auxiliaires pour manipuler les structures de modules du MiniML
- Le module `Extraction` est le coeur de l'extraction. C'est là qu'un terme Coq est traduit en un terme ou un type MiniML.
- Les trois modules `Ocaml`, `Haskell` et `Scheme` servent chacun à traduire les objets MiniML vers la syntaxe concrète d'un des langages cibles.
- Le module `Common` factorise un certain nombre de fonctions de renommage entre les trois langages-cibles.
- Enfin, `Extact_env` permet de déterminer quel environnement minimal d'objets Coq devra être extrait pour satisfaire la requête d'un utilisateur.

4.4.2 Petit manuel utilisateur

Naturellement, ce petit manuel sera moins détaillé que le chapitre de référence présent dans le Manuel de Référence Coq [78]. Mais le présent document ayant vocation à être

un tour d'horizon aussi complet que possible de l'extraction en Coq, voici donc un aperçu succinct des commandes Coq concernant l'extraction.

L'extraction proprement dite

```
Coq < Extraction plus.
Coq < Recursive Extraction plus minus.
Coq < Extraction "monfichier" plus minus.
```

La première commande est la plus simple, elle ne fait qu'afficher à l'écran l'extraction d'un objet, ici `plus`. La seconde, par contre, affiche également tout ce dont dépend le ou les objet(s) demandé(s). Dans notre exemple, on verra également s'afficher l'extraction du type `nat`. Enfin la troisième commande se comporte exactement comme la seconde, sauf qu'elle écrit dans un fichier et non sur l'écran. Normalement, étant donné que ce fichier contient alors toutes les dépendances nécessaires, il est directement compilable. Notons que si le langage-cible choisi est Ocaml, on obtient à la fois un fichier *.ml et une interface *.mli. Enfin, ces commandes acceptent également un module `M` à la place d'un objet comme `plus`.

```
Coq < Extraction Library Peano.
Coq < Recursive Extraction Library Peano.
```

Ces deux commandes quelque peu obsolètes sont destinées à extraire d'un coup tout le contenu d'une bibliothèque Coq, c'est à dire d'un fichier *.v compilé et chargé via un `Require`. Dans le cas de la bibliothèque `Peano.v`, on obtient alors `peano.ml` et `peano.mli`. Dans la deuxième variante, l'extraction produit également des fichiers *.ml et *.mli pour toutes les bibliothèques dont dépend `Peano.v`.

Contrôler l'extraction

On peut tout d'abord modifier le langage-cible :

```
Coq < Extraction Language Ocaml.
Coq < Extraction Language Haskell.
Coq < Extraction Language Scheme.
```

On retrouve ensuite les commandes de contrôle des « optimisations », déjà rencontrées dans la section 4.3 précédente :

```
Coq < Set Extraction Flag n.
Coq < Set Extraction Optimize.
Coq < Unset Extraction Optimize.
```

Compte tenu de l'état actuel de ces optimisations, il est parfois préférable d'utiliser pour l'instant la version `Unset`.

Voici enfin les commandes contrôlant le dépliage de fonctions, déjà décrites également dans la section 4.3 précédente :

```
Coq < Unset Extraction AutoInline.
Coq < Set Extraction AutoInline.
Coq < Extraction Inline f g.
Coq < Extraction NoInline h k.
Coq < Reset Extraction Inline.
```

Les axiomes

Il est en fait possible d'utiliser l'extraction lorsque l'on est en présence d'axiomes dans un développement. Si un axiome logique est utilisé, l'extraction se contente d'afficher un message d'avertissement rappelant que la correction du code extrait va alors dépendre de la validité de cet axiome logique. Si maintenant le développement utilise un axiome informatif `f`, le code extrait qui en dépend contiendra alors une exception signalant du code manquant devant être réalisé :

```
let f = failwith "AXIOM TO BE REALIZED"
```

L'utilisateur doit alors fournir du code réalisant effectivement cet axiome. Ceci peut être fait manuellement sur le fichier extrait, ou bien via une commande spéciale :

```
Coq < Extract Constant f ⇒ "mon_code_realisant".
```

Cette commande n'est qu'une commodité syntaxique, elle ne vérifie pas le contenu de la chaîne de caractères fournie en réalisation de l'axiome. Elle engendre alors :

```
let f = mon_code_realisant
```

Il se peut également qu'un axiome soit un schéma de types. Par exemple, pour `t : Set` → `Set` → `Set` , la réalisation suivante :

```
Coq < Extract Constant t "'a" "'b" ⇒ "'a * 'b".
```

engendre la déclaration de type :

```
type ('a,'b) t = 'a * 'b
```

Même s'il ne s'agit plus d'axiomes, notons enfin la possibilité de remplacer un type inductif par un autre au moment de l'extraction. Ainsi le type `Coq sumbool` avec ses deux constructeurs `left` et `right`, présenté p.29, est isomorphe à `bool` après extraction. On peut alors identifier ces deux types ainsi :

```
Coq < Extract Constant sumbool ⇒ "bool" [ "true" "false" ].
```

4.5 Un premier exemple complet

Nous allons maintenant donner comme illustration l'extraction de la fonction `div` de la page 31. Cette fonction, avec pré- et post-conditions, est définie en utilisant la bonne

fondation de l'ordre usuel sur les entiers de Peano. Si l'on extrait la fonction nommée `well_founded_induction` utilisée dans la définition par preuve de notre fonction `div`, on obtient un combinateur de point-fixe à la `Y`, sans trace de parties logiques mis à part un lambda anonyme :

```
let rec well_founded_induction f a =
  f a (fun y _ → well_founded_induction f y)
```

Et cette constante `well_founded_induction` va être dépliée automatiquement dans `div`, laissant au final le point-fixe non structurel voulu. D'autre part `div` utilise une fonction auxiliaire `le_lt_dec` servant à comparer deux entiers. Cette fonction, voisine de la fonction `lt_eq_lt_dec` déjà rencontrée, travaille avec des objets de type `sumbool`. Comme signalé juste avant, il peut être agréable de remplacer `sumbool` par des booléens :

```
Coq < Extract Inductive sumbool ⇒ bool [ true false ].
```

Il ne reste plus alors qu'à extraire :

```
Coq < Extraction "div.ml" div.
```

L'interface `div.mli` obtenue est alors :

```
type nat =
  | O
  | S of nat

type 'a sig0 = 'a
  (* singleton inductive, whose constructor was exist *)

val minus : nat → nat → nat

val le_lt_dec : nat → nat → bool

val div : nat → nat → nat sig0
```

Dans cette signature, le `sig0` n'est plus qu'un alias rappelant qu'auparavant le type de `div` finissait par une existentielle. Mis à part cela, on obtient bien le type `nat→nat→nat` attendu pour `div`.

Voici enfin le fichier `div.ml` :

```
type nat =
  | O
  | S of nat

type 'a sig0 = 'a
  (* singleton inductive, whose constructor was exist *)

(** val minus : nat → nat → nat **)
```

.../...

…/…

```
let rec minus n m =
  match n with
    | O → O
    | S k → (match m with
                | O → S k
                | S l → minus k l)

(** val le_lt_dec : nat → nat → bool **)

let rec le_lt_dec n m =
  match n with
    | O → true
    | S n0 → (match m with
                 | O → false
                 | S n1 → le_lt_dec n0 n1)

(** val div : nat → nat → nat sig0 **)

let rec div x b =
  match le_lt_dec b x with
    | true → S (div (minus x b) b)
    | false → O
```

Chapitre 5

Exemples d'extraction

Depuis le début des travaux sur l'extraction en Coq par C. Paulin, il faut bien reconnaître qu'aucune application d'échelle industrielle n'a été certifiée via cette méthode. Ceci dit, un certain nombre d'exemples de taille significative ont été réalisés. Nous allons ici tenter de dresser un panorama succinct de ces exemples. Parmi ces exemples, nous en avons sélectionné quatre qui ont tiré bénéfice de notre nouvelle extraction ou bien n'auraient pas pu exister sans elle. Ces cas pratiques que nous avons choisis d'illustrer sont :

- la contribution `Lannion` de J.-F. Monin,
- la contribution `Rocq/HIGMAN` de H. Herbelin,
- la contribution `Nijmegen/C-CoRN` de l'équipe de H. Barendregt,
- la contribution `Rocq/FSets` par J.-F. Filliâtre et l'auteur.

Les deux premières contributions sont détaillées dans ce chapitre. La troisième fait l'objet du chapitre suivant, dédié à l'extraction de formalisations constructives de nombres réels. Enfin, la quatrième contribution est décrite dans le chapitre 7.

5.1 La bibliothèque standard de Coq

Si l'on recherche des développements Coq en quête de preuves à extraire, la première source accessible est la bibliothèque standard de Coq, qui est directement fournie avec le système. Bon nombre des fonctions données en illustration jusqu'ici proviennent de là ou en sont dérivées. Nous avons mis en place un test consistant à extraire systématiquement tout le contenu de cette bibliothèque standard. Ce test se trouve dans le répertoire `contrib/extraction/test` des sources de Coq. Une fois dans ce répertoire, taper `make tree; make` permet de lancer le test.

Quand on regarde le détail des fonctions extraites à partir de cette bibliothèque standard, le bilan est maigre. Tout d'abord, la plupart des résultats de cette bibliothèque sont des propriétés logiques, et n'apparaissent donc pas à l'extraction. Ensuite, la plus importante composante de cette bibliothèque, à savoir les `Reals`, sont hors du champ de l'extraction. En effet il s'agit d'une formalisation axiomatique des nombres réels, commençant par :

```
Parameter R : Set.
Parameter R0 : R. (* one *)
Parameter R1 : R. (* zero *)
Parameter Rplus : R → R → R.
```

Ceci n'est pas forcément rédhibitoire pour l'extraction, qui peut travailler avec des axiomes comme on vient de le voir dans le chapitre précédent. Mais certains de ces axiomes correspondent à une vision non pas constructive mais classique des réels, et ne peuvent donc pas être réalisés fidèlement. C'est le cas par exemple de :

```
Axiom total_order_T : ∀r1 r2:R, {r1 < r2} + {r1 = r2} + {r1 > r2}.
```

On sait en effet qu'aucune arithmétique réelle exacte ne peut avoir d'égalité décidable. Plus gênant encore, le dernier des axiomes, `completeness`, demande l'existence constructive d'une plus petite borne supérieure pour tout ensemble de réels non vide et borné supérieurement. Or un ensemble est ici un objet logique de type `R→Prop`. Cet axiome est donc non réalisable.

Au final, il ne reste après extraction que des fonctions élémentaires concernant des structures de données comme les entiers de Peano, les listes, les entiers binaires. Ces derniers constituent la partie la plus intéressante de ces exemples, avec ainsi des fonctions de divisions, de racines carrées, etc. On trouve également un algorithme de tri de listes dans le répertoire `theories/Sorting`. Enfin le répertoire `theories/IntMap` contient une formalisation d'ensembles finis indexés par des entiers. Sur l'ensemble de ces fonctions informatives, on ne trouve finalement qu'assez peu de fonctions définies par tactiques, les seules vraiment intéressantes du point de vue extraction. Et l'extraction se comporte raisonnablement bien sur ces fonctions, qui sont toutes suffisamment standard pour ne pas avoir besoin de `Obj.magic` pour typer.

5.2 Les contributions utilisateurs

La principale base d'exemples d'extraction se situe aujourd'hui dans les développements soumis par les utilisateurs de Coq. Ces *contributions utilisateurs* sont regroupées et classées par l'équipe de développement de Coq (par origine géographique et par thème). Elles sont également maintenues à jour à chaque modification de Coq et publiées à chaque nouvelle version de Coq. Il est possible de consulter la liste des 88 contributions actuelles sur le site `http://coq.inria.fr/contribs-eng.html`.

Évidemment, cette base de données de développements Coq n'est en rien exhaustive, puisqu'elle repose sur une déclaration de la part des utilisateurs. Et certains développements Coq fort intéressants vis-à-vis de l'extraction ne font pas partie pour l'instant de ces contributions soumises par les utilisateurs. Parmi de tels développements, on peut citer par exemple une procédure de décision dédiée à la logique propositionnelle intuitionniste, par K. Weich [82], ou bien encore un analyseur statique de programmes par interprétation abstraite, par D. Cachera, T. Jensen, D. Pichardie et V. Rusu [17]. Et il existe très certainement d'autres travaux qui mériteraient d'être cités ici mais dont nous n'avons pas encore

pris connaissance.

Revenons maintenant à ces contributions. Déjà, beaucoup ne concernent pas l'extraction, car elles établissent uniquement des résultats dans `Prop`. Néanmoins, une trentaine d'entre elles sont pertinentes pour l'extraction. La liste complète de ces contributions extractibles se trouve en Annexe A.

Si l'on cherche à classifier quelque peu ces multiples exemples, on peut tout d'abord distinguer un certain nombre de procédures de décision. On en trouve ainsi plusieurs concernant les formules booléennes : `Dyade/BDDS`, `Suresnes/BDD` et `Sophia-Antipolis/Stalmarck` [56]. Quant à `Rocq/GRAPHS`, elle traite d'(in)égalités linéaires sur Z. `Nancy/FOUnify` et `Lannion` résolvent le problème de l'unification de termes du premier ordre. Et `Rocq/COC` propose un vérificateur de types pour le calcul des constructions.

D'autre part, plusieurs contributions sont centrées sur des structures de données, et peuvent donc servir de bibliothèque de base pour d'autres travaux. On trouve ainsi des définitions des nombres rationnels, `Nijmegen/QArith` et `Orsay/QArith`, des nombres réels constructifs dans `Nijmegen/C-CoRn`, des structures d'arbres dans `Bordeaux/SearchTrees`, `Bordeaux/dictionnaries` et `Orsay/FSets`. Enfin, les formalisations de BDD déjà citées peuvent également entrer dans cette catégorie.

Par contre, la grande variété des contributions empêche de pousser beaucoup plus loin une telle classification. Et il n'y a pas que les sujets d'étude qui varient grandement dans ces contributions. Tout d'abord, la taille de ces contributions peut aller d'un simple fichier de 300 lignes, comme pour `Bordeaux/Exceptions`, jusqu'à plus d'une centaine de fichiers et 75 000 lignes pour `Nijmegen/C-CoRN`.

L'âge est également très variable. Parmi les contributions les plus anciennes, on trouve `Rocq/Higman` fait vers 1992 avec Coq version 5.x. À l'opposé, certaines contributions tirent parti de fonctionnalités récentes de Coq, comme les nouveaux modules pour `Orsay/FSets` et `Bordeaux/dictionnaries`.

L'attitude des auteurs vis-à-vis de l'extraction diffère également selon les contributions. Dans certains cas, les auteurs avaient prévu dès le départ d'extraire leurs résultats. Par exemple, B. Barras avait intégré au `Makefile` de `Rocq/COC` l'extraction, la compilation du fichier extrait et le lancement d'un test du programme extrait. À l'opposé, les développeurs de `Nijmegen/C-CoRN`, bien que conscients de la possibilité théorique d'extraire leurs travaux, ne l'avaient pas envisagé à l'origine. Dans beaucoup d'autres contributions, nous avons mis en valeur les aspects calculatoires, en ajoutant des extractions, compilations et tests automatiques dans l'esprit de `Rocq/COC`. Ceci a été fait lors d'un inventaire manuel des contributions, peut-être reste-t-il encore des exemples extractibles non repérés à l'heure actuelle.

Enfin, la difficulté d'extraction varie grandement. Ainsi dans `Lyon/Firing-Squad`, le principal objet extrait est une fonction de transition purement informative, écrite via un `Fixpoint`. L'extraction consiste alors en une simple traduction vers la syntaxe du langage cible. Le programme finalement obtenu est intéressant en soit car il permet grâce à une petite interface graphique de visualiser l'évolution des états sous l'action de cette fonction de transition, mais l'intérêt du pur point de vue extraction est très réduit. Au passage, notons

qu'on rencontre là un usage annexe de l'extraction peu mis en avant, à savoir l'utilisation à des fins de simulation. Plus généralement, on s'aperçoit qu'il y a peu d'extractions de fonctions définies par tactiques. D'autre part la plupart des cas fonctionnaient ou auraient pu fonctionner avec l'ancienne extraction.

Tous les points précédents font qu'il reste peu de contributions permettant d'étudier les apports de notre nouvelle extraction. Il y a ainsi seulement quatre contributions qui s'extraient vers du code mal typé, nécessitant le recours à des `Obj.magic` : Il s'agit de `Lyon/Circuits`, `Lannion`, `Nijmegen/C-CoRN` et `Rocq/Higman`. Et nous avons déjà signalé qu'uniquement deux contributions utilisent le nouveau système de modules. Enfin, nous n'avons testé notre extraction des types co-inductifs vers Ocaml que dans la contribution nommée `Rocq/Mutual-Exclusion`. Nous allons maintenant étudier plus en détail certaines de ces contributions tirant parti de notre nouvelle extraction.

5.3 Exceptions par continuations en Coq

Il s'agit de travaux de J.-F. Monin concernant la certification de programmes fonctionnels avec exceptions. Ces travaux [60, 61] ont été effectués entre 1995 et 1997. Historiquement, il s'agit d'une des deux premières situations concrètes où l'ancienne extraction de Coq a généré du code non typable. La deuxième de ces situations concrètes correspond à la contribution `Rocq/Higman` que nous étudierons juste après.

5.3.1 Modélisation des exceptions en Coq

Pour modéliser en Coq des exceptions, J.-F. Monin utilise une traduction par continuation (CPS, pour « continuation passing style »). Nous allons tout de suite entrer dans le vif du sujet, le lecteur désireux de trouver une introduction plus graduelle à ces travaux pourra se reporter à [61]. Voici tout d'abord le schéma de types utilisé pour exprimer le type résultat d'une fonction pouvant lever une exception :

```
Definition Mx (C:Prop)(A:Set) :=
  ∀(X:Set)(P:Prop),(P→X) → (C → P) → (A → X) → X.
```

Ici, le type informatif `A` est le type de retour « normal » de la fonction que l'on va considérer, alors que `C` est une condition logique entraînant la levée d'une exception. La présence de la quantification sur `X` correspond à la traduction par continuation[1]. Enfin, `P` et l'argument suivant `e` de type `P→X` représentent la manière de construire le résultat `X` de la continuation en cas d'exception. Au final, le type `(Mx C A)` peut être vu comme un type somme `A∨C`, représentant les deux issues possibles, standard ou exceptionnelle, de notre fonction.

Voyons maintenant comment fabriquer des valeurs de ce type :

```
Definition Mx_unit (C:Prop)(A:Set)(a:A) : Mx C A :=
```

…/…

[1]Sans les exceptions, cette traduction aurait consisté à passer de `A` à `∀X,(A→X)→X`.

.../...

```
  fun X P e i k ⇒ k a.

Definition Mx_raise (C:Prop)(A:Set)(c:C) : Mx C A :=
  fun X P e i k ⇒ e (i c).
```

La première fonction correspond au résultat normal : si l'on a réussi à bâtir un objet `a` de type `A`, il suffit de l'appliquer à la continuation courante `k` de type `A→X`. À l'inverse, la deuxième fonction traite la levée d'une exception : si l'on a obtenu la preuve `c` que l'on est dans la situation exceptionnelle `C`, on utilise le mécanisme de génération des exceptions, à savoir les fonctions `i` puis `e` de types respectifs `C→P` et `P→X`.

Voici maintenant deux fonctions permettant de traiter les exceptions. La première correspond à une analyse par cas :

```
Definition Mx_try (C:Prop)(A:Set)(m:Mx C A)(X:Set)(k:A→X)(e:C→X): X :=
  m X C e (fun p ⇒ p) k.
```

Autrement dit, si l'on sait toujours engendrer un résultat de type `X` aussi bien à partir d'une valeur normale de type `A` que d'une exception de type `C`, alors on peut toujours bâtir un résultat dans `X` à partir d'un objet `m` dans le type somme `(Mx C A)`. D'autre part, on peut passer d'un type avec exceptions à un autre type avec exceptions par le biais de la fonction suivante :

```
Definition Mx_bind
  (A A':Set)(C C':Prop)(m:Mx C A)(f:A→Mx C' A')(j:C→C') : Mx C' A' :=
  fun X P e i k ⇒ m X P e (fun c ⇒ i (j c)) (fun a ⇒ f a X P e i k).
```

Cet opérateur permet en pratique de composer deux fonctions pouvant chacune lever des exceptions.

5.3.2 Imprédicativité et typage des fonctions extraites

La contribution `Lannion` utilise l'imprédicativité de `Set` (cf. page 23) au niveau du type `Mx`. En effet, ce type, qui contient une quantification universelle sur `X:Set`, doit a priori être dans `Type`. Mais si l'on active l'imprédicativité de `Set`, `Mx` peut alors être de type `Set`. Ceci permet alors de prendre `X=(Mx C A)` à l'intérieur de `(Mx C A)`. Cette situation se produit par exemple lorsqu'on construit un résultat de type `Mx C A` via un `Mx_try`. Une telle situation imprédicative est un indice fort de la présence d'erreurs de typage dans le code extrait brut. J.-F. Monin remarque d'ailleurs p.48 de [61] que cela se produit pour les points-fixes (traduits en continuations) du genre :

```
let rec f x = ... try ... f y ... with ...
```

Nous verrons par la suite que l'imprédicativité seule n'entraîne pas nécessairement le besoin d'`Obj.magic` dans le code extrait, et que des `Obj.magic` peuvent apparaître en l'absence d'objets imprédicatifs. En tout cas, historiquement, le besoin d'`Obj.magic` dans le code extrait s'est fait sentir en premier lieu dans `Lannion` et `Rocq/Higman`, deux contributions utilisant l'imprédicativité, ce qui n'est pas un hasard.

5.3.3 L'extraction de cette modélisation

Si l'on s'en tient à l'extraction des types exposée au chapitre 3, le type `Mx` est traduit de façon très approchée, à cause de ses deux quantifications universelles, la deuxième étant d'ailleurs logique :

```
type 'a mx = __ → __ → (__ → __) → __ → ('a → __) → __
```

Par contre, dans le type de `Mx_unit` et `Mx_raise`, ce schéma de types `Mx` est utilisé en position de tête, et ses quantifications universelles se retrouvent alors au premier niveau. L'amélioration des types décrite à la section 3.5.4 donne alors un type beaucoup plus standard à ces deux fonctions :

```
(** val mx_unit : 'a1 → (__ → 'a2) → ('a1 → 'a2) → 'a2 **)

let mx_unit a e k = k a

(** val mx_raise : (__ → 'a2) → ('a1 → 'a2) → 'a2 **)

let mx_raise e k = e __
```

On constate en particulier que l'extraction de ces deux fonctions ne nécessite pas l'utilisation de `Obj.magic`, même si leur utilisation ultérieure peut en demander. Par contre la situation se corse en ce qui concerne `Mx_try`, puisqu'on trouve alors `Mx` dans le type d'un des arguments de cette fonction, et non plus en position de tête.

```
(** val mx_try : 'a1 mx → ('a1 → 'a2) → (__ → 'a2) → 'a2 **)

let mx_try m k e = Obj.magic m __ __ e __ k
```

On voit donc apparaître des `Obj.magic`. Par exemple, l'argument `e` de `Mx_try` a pour type `C→X` au niveau Coq, qui devient `__→'a2` au niveau Ocaml puisque `C` est logique et que l'on tente de faire de `X` une variable de type selon l'amélioration 3.5.4. Mais l'objet `m` de type `(Mx C A)` attend au niveau Ocaml un troisième argument de type `__→__`. L'usage d'un `Obj.magic` est donc nécessaire si l'on souhaite garder la généricité du type de `e`. En fait, si l'on n'appliquait pas l'amélioration 3.5.4, le type de `mx_try` serait composé essentiellement de `__`, et cette fonction serait typable sans `Obj.magic`. Mais procéder ainsi ne ferait que repousser le problème plus loin, lors de l'utilisation de `mx_try`.

Enfin, pour `Mx_bind`, la situation est similaire à celle de `Mx_try` :

```
(** val mx_bind :
    'a1 mx → ('a1 → 'a2 mx) → (__ → 'a3) → ('a2 → 'a3) → 'a3 **)

let mx_bind m f e k =
  Obj.magic m __ __ e __ (fun a → Obj.magic f a __ __ e __ k)
```

5.3.4 Utilisation de ces exceptions

La contribution `Lannion` contient un certain nombre d'applications de ces exceptions à des cas pratiques. Nous allons maintenant étudier l'extraction de ces différents cas.

Unification de termes du premier ordre

Dans le répertoire `Lannion/continuations/FOUnify_cps`, J.-F. Monin reprend l'étude de J. Rouyer d'une procédure de décision de l'unification de termes du premier ordre (cf. `Nancy/FOUnify`). Simplement, en cas d'échec de l'unification de deux sous-termes, une exception est désormais levée au lieu de finir de traiter tous les calculs restants.

Dans ce cas simple (une seule exception, sans contenu), J.-F. Monin utilise en fait une version simplifiée des exceptions présentées précédemment, sans le polymorphisme lié au ∀`X`. Au lieu de `Mx`, le type des exceptions est :

```
Definition Nx (X:Set)(P:Prop)(e:P→X)(C:Prop)(A:Set) :=
  (C → P) → (A → X) → X.
```

Et le fait d'avoir `X`, `P` et `e` en paramètres plutôt qu'en variables universellement quantifiées améliore grandement la fidélité du type extrait :

```
type ('x,'a) nx = __ → ('a → 'x) → 'x
```

Et le dernier `__` restant correspond à l'argument logique de type `C→P`. Au final, le programme d'unification extrait est directement typable sans `Obj.magic`, ce que J.-F. Monin avait déjà constaté dans [61] :

> *Cet exemple ne comporte qu'un seul* `try ... with` *qui se trouve à l'extérieur de l'appel à une fonction récursive. L'imprédicativité de* `Mx` *n'est donc pas mise à contribution, le programme extrait est typable en ML.*

Et notre extraction n'engendre ici aucun `Obj.magic` superflu.

Parcours d'arbres avec exceptions

Il s'agit des exemples du répertoire `Lannion/continuations/weight`. Le but y est de calculer le « poids » d'un arbre binaire, à savoir la somme des entiers présents à ses feuilles. Et l'usage d'exception est introduit progressivement afin de répondre aux contraintes (arbitraires) suivantes :

- On arrête le parcours si la somme partielle courante dépasse une certaine quantité prédéfinie.
- On arrête également le parcours si l'on rencontre un zéro dans l'une des feuilles.

En pratique, ces exemples utilisent des exceptions du même genre que l'exemple d'unification précédent, c'est à dire avec un paramètre `X` fixé à l'avance et non pas une quantification universelle sur `X`. À l'extraction, on constate que ces exemples sont plus complexes que celui de l'unification : des `Obj.magic` apparaissent en effet. Mais ces `Obj.magic` servent ici juste à assurer la conformité du type de chaque sous-terme avec ce qu'a décidé notre extraction des types. Et en l'occurrence le code extrait reste ici typable si l'on enlève ces `Obj.magic`.

L'algorithme d'Huffman

Le répertoire `Lannion/polycont` contient le premier exemple utilisant la situation imprédicative identifiée par J.-F. Monin, à savoir la gestion d'exceptions dans une boucle récursive.

Pour mémoire, l'algorithme d'Huffman, dans sa partie encodage, associe un chemin[2] dans un arbre `t` à un objet `a` que l'on souhaite encoder. En voici une version codée manuellement :

```
let encode a t =
  let rec lookup = function
    | Leaf → raise Not_found
    | Node(t1,b,t2) →
       if a=b then []
       else try L::lookup t1 with Not_found → R::lookup t2
  in lookup t
```

Et voici la version finalement extraite à partir de la preuve Coq de J.-F. Monin :

```
(** val lookup : 'a1 → 'a1 tree → (__ → 'a2) →
                 (direction list sig0 → 'a2) → 'a2 **)

let lookup a t e x =
  let rec lookup0 = function
    | Leaf → (fun _ _ x0 _ x1 → x0 __)
    | Node (t1, b, t2) →
        (match eg a b with
           | true → (fun _ _ x0 _ x1 → x1 Nil)
           | false →
               mx_try (Obj.magic lookup0 t1) (fun h _ _ x0 _ x1 →
                 x1 (Cons (L, h))) (fun _ _ _ x0 _ x1 →
                 mx_bind (Obj.magic lookup0 t2) (fun l2 _ _ x2 _ x3 →
                   x3 (Cons (R, l2))) x0 x1))
  in lookup0 t __ __ e __ x

(** val encode : 'a1 → 'a1 tree → direction list sig0 **)

let encode a t =
  mx_try (fun _ _ x _ x0 → lookup a t x x0) (fun x → x) (fun _ → Nil)
```

On constate donc que cette version est très complexe, et contient deux `Obj.magic` (plus les trois dissimulés sous `mx_try` et `mx_bind`). En fait, comme l'avait remarqué J.-F. Monin, un seul `Obj.magic` devant le `mx_try` est suffisant.

D'autre part, on constate que la présence des `Obj.magic` gêne les optimisations effectuées normalement par l'extraction, qui permettent habituellement de faire disparaître la plupart des résidus logiques `_` et `__`. Nous comptons améliorer cela à l'avenir.

En tout cas, le code généré, sans être élégant, présente l'avantage d'être engendré automatiquement, `Obj.magic` compris, et de pouvoir être compilé directement.

[2] c'est-à-dire une liste de directions `L` ou `R`

Partage maximal de sous-termes communs

Nous ne développerons pas le dernier exemple de cette contribution, à savoir une transformation d'arbre avec partage maximal des sous-arbres. En effet le comportement de cet exemple vis-à-vis de l'extraction est similaire à celui de Huffman.

5.4 Le lemme de Higman

Nous allons maintenant étudier l'extraction d'une autre contribution faisant intervenir l'imprédicativité, à savoir `Rocq/Higman`. Il s'agit d'une preuve constructive du lemme de Higman pour le cas des mots sur un alphabet à deux lettres, implanté en Coq par H. Herbelin vers 1992.

Le lemme de Higman est un résultat combinatoire qui stipule que pour toute suite w_n de mots sur un alphabet fini, il existe deux indices $i < j$ tels que w_i est un sous-mot de w_j, au sens où il suffit d'enlever certaines lettres de w_j pour obtenir w_i, ce que nous noterons alors $w_i \trianglelefteq w_j$. Ce lemme a également une formulation en terme de quasi-ordres bien fondés, sur des alphabets qui ne sont alors plus nécessairement finis. Comme notre alphabet n'a ici que deux lettres, nous n'utiliserons pas cette formulation généralisée.

Ce résultat a donné lieu à de très nombreux travaux dans la communauté intuitionniste, dans le but d'élaborer une preuve constructive dont l'algorithme dérivé par extraction soit le plus efficace possible. Au lieu de dresser ici l'historique complet de ces travaux, nous préférons renvoyer à la très bonne étude de M. Seisenberger [74], et en particulier au tableau synoptique p.47. Pour replacer la formalisation de H. Herbelin dans son contexte, précisons juste qu'une preuve classique élégante de ce lemme de Higman a été trouvée par Nash-Williams dans les années 60, soit une dizaine d'années après les travaux de Higman. Cette preuve est basée sur un argument dit de « plus petite mauvaise séquence ». Puis il a fallu attendre 1990 pour voir C. Murthy formaliser cette preuve classique en Nuprl puis en obtenir une version constructive via le procédé de A-traduction de Friedman. Indépendemment, H. Herbelin a formalisé vers 1992 une A-traduction « par nécessité » en Coq, introduisant un minimum de doubles-négations.

Mais il est vite apparu que l'algorithme correspondant à cette preuve constructive initiale a de très mauvaises propriétés calculatoires, comme nous allons le vérifier par la suite. C'est pourquoi de nombreuses autres preuves constructives de ce lemme ont été proposées ultérieurement, comme par exemple [63] ou [21]. D'après H. Herbelin :

> *La motivation de T. Coquand n'était pas tant l'extraction, mais la compréhension de la part d'imprédicativité nécessaire à la preuve, en l'occurrence l'induction sur des arbres à branchement infini, ainsi que la symétrisation de la preuve, c'est-à-dire s'abstraire de bien ordonner l'alphabet. Je devrais aussi ajouter son besoin très fort d'élégance, et son souci de dégager l'essence des théorèmes mathématiques.*

La preuve de [21] est en fait une reformulation de celle de Nash-Williams, dans laquelle le raisonnement par l'absurde a été rendu positif via l'utilisation de types inductifs ad hoc.

Cette preuve proposée par T. Coquand et D. Fridlender a depuis été formalisée en Minlog par M. Seisenberger [74] et en Isabelle par S. Berghofer [13, 14]. Enfin, S. Berghofer a récemment adapté sa preuve d'Isabelle vers Coq. Cette nouvelle formalisation en Coq du lemme de Higman, qui est désormais une contribution utilisateur nommée `Muenchen/Higman`, est particulièrement concise tout en fournissant un code extrait excessivement plus efficace que le code extrait de `Rocq/Higman`.

5.4.1 Higman et l'imprédicativité

Nous allons tout d'abord illustrer le mécanisme de la démonstration formalisée par H. Herbelin, qui est donc à la base due à Nash-Williams pour sa partie classique, puis transformée en preuve constructive par A-traduction. En particulier nous allons voir comment cette formalisation fait intervenir l'imprédicativité, ce qui entraîne lors de l'extraction la présence de types non traduisibles en ML.

La preuve se fait en raisonnant sur une hypothétique « plus petite mauvaise séquence ». Dans ce contexte, une « mauvaise séquence » est une suite de mots qui invaliderait le lemme de Higman. Ici les suites sont formalisées via des relations[3] informatives (*i.e.* sur `Set`) reliant entiers et mots :

```
Definition seq := nat → word → Set.
Section Bad_sequence.
 Variable f : seq.
 Definition exi_im := ∀n, (∀x, f n x → A) → A.
 Definition uniq_im := ∀n x y, f n x → f n y → (x = y → A) → A.
 Definition cex := ∀i j x y, f i x → f j y → i < j → x ≤ y → A.
 Inductive bad : Set := bad_intro : exi_im → uniq_im → cex → bad.
End Bad_sequence.
```

Les propriétés `exi_im` et `uniq_im` stipulent que notre relation `f` est en fait fonctionnelle, en demandant respectivement l'existence et l'unicité de l'image d'un entier `n` par `f`. A ce propos, on notera les effets de la A-traduction. Intuitivement, `A` joue le rôle de `False`, mais servira au final à contenir le résultat de l'algorithme constructif. Par exemple, `(∀x, f n x → A)→A` se lit informellement comme ¬(∀x,¬(f n x)), ce qui est classiquement équivalent à `∃x, f n x`. Ensuite, la propriété `cex` exprime le fait que `f` invalide bien le lemme de Higman. Et finalement, une séquence `f` est « mauvaise » si l'on a `(bad f)`, c'est-à-dire que les trois conditions précédentes sont réunies.

Une fois précisé ce qu'est une « mauvaise séquence », passons maintenant à la définition de « plus petite mauvaise séquence » :

[3] H. Herbelin nous a signalé que cet usage de relations, impliquant l'imprédicativité, permet d'éviter le recours à l'axiome de description nécessaire à la preuve via des fonctions.

```
(* To be equal on the n-1 first terms *)

Definition eqgn (n : nat) (h h' : seq) :=
  ∀i, i < n → ∀s t, h i s → h' i t → (s = t → A) → A.

(* To be minimal on the nth term *)

Definition Minbad (n : nat) (h : seq) (y : word) :=
  ∀h' : seq, bad h' → eqgn n h h' → ∀z, h' n z → z ≺ y → A.

(* To be minimal on the first n-1 terms *)

Definition Minbadns (n : nat) (h : seq) :=
  ∀p, p < n → (∀y, h p y → Minbad p h y → A) → A.

(* The minimal (bad) counter-example *)

Definition Mincex (n : nat) (x : word) :=
  ∀C : Set,
   (∀h : seq, bad h → Minbadns n h → h n x → Minbad n h x → C) → C.
```

Nous allons nous attarder essentiellement sur la dernière définition. On y reconnaît en fait l'encodage imprédicatif d'une quantification universelle, `Mincex` signifiant informellement :

```
Definition Mincex (n : nat) (x : word) :=
  ∃h:seq, bad h ∧ Minbadns n h ∧ h n x ∧ Minbad n h x.
```

On peut également interpréter `Mincex` comme une union infinie portant sur toutes les séquences. Et le caractère imprédicatif de cet encodage provient bien sûr de la quantification sur tous les types `C:Type`, alors même que l'on souhaite que `Mincex` soit une séquence, et donc ait un résultat dans `Set`.

Après avoir ainsi bâti cette plus petite mauvaise séquence de façon abstraite, la preuve établit ensuite que, s'il existe une mauvaise séquence (pas forcément minimale), alors `Mincex` est bien une également une mauvaise séquence (*i.e.* on a `bad Mincex`). Puis il montre comment dériver de toute mauvaise séquence une autre mauvaise séquence plus petite. En examinant la séquence ainsi dérivée à partir de `Mincex`, on obtient alors une contradiction (c'est-à-dire une preuve de `A`), compte tenu des propriétés de minimalité de `Mincex`. Il ne reste plus qu'à choisir soigneusement `A` pour conclure, ici `∃i j, i<j → f i ⊴ f j`.

5.4.2 L'extraction de Higman

Historiquement, cette formalisation a été extraite par H. Herbelin vers 1992, en utilisant une version 5.x de Coq. Il s'agissait alors de l'une des premières versions de l'extraction en Coq, implantée alors par C. Paulin et B. Werner. Les langages cibles de l'extraction lors de ces premières expérimentations était Caml Lourd et aussi LazyML, une version paresseuse de ML, que l'on retrouve aujourd'hui encore comme base du compilateur hbc pour Haskell.

Outre l'attrait de la paresse proche de la stratégie de réduction Coq, LazyML présentait

également comme intérêt d'avoir à l'époque une option de compilation désactivant complètement la vérification du typage.

En fait, nous avons constaté qu'il n'y a qu'un seul emplacement dans tout le code extrait qui nécessite un contournement du typage ML. Ceci est plutôt surprenant étant donné le rôle central de `Mincex` dans la formalisation et son type incompatible avec ML. L'extraction actuelle retrouve bien automatiquement ce point d'achoppement, et y insère un `Obj.magic` :

```
let snake badf f0 f_0_f0 a h n h0 h1 =
  xa_elim a n h1 (fun y h3 →
    Obj.magic h3 __ (fun _ h5 h6 h7 h8 →
      h8 __ (badfx (badMin badf f0 f_0_f0) a h n (Leastp_intro (h1, h0)))
        (fun x _ x0 x2 x3 x4 x5 →
        eqgn_fxMin_h badf f0 f_0_f0 a n h5 h6 x x0 x2 x3 x4 x5) y (Fxtl
        (n, (Leastp_intro (h1, h0)), h3)) __))
```

Par contre, tout comme pour Lannion, d'autres `Obj.magic` « superflus » sont également ajoutés. Ils servent à faire coïncider le type approximatif choisi pour `Mincex` et les types des fonctions manipulant de tels objets `Mincex`. Si l'objectif est uniquement de compiler le fichier `higman.ml` créé par l'extraction, alors ces 14 `Obj.magic` supplémentaires peuvent être enlevés. Mais le fichier alors obtenu n'est plus compatible avec l'interface `higman.mli` automatiquement générée.

Revenons maintenant à l'étude du programme obtenu. Le code extrait est relativement court (435 lignes) mais incompréhensible même une bonne connaissance préalable de la preuve d'origine, comme l'illustre l'extrait ci-dessus. Quant à l'exécution de ce programme, elle est possible, du moins pour de tous petits exemples :

```
./higman 101 0110 01010
f(0)=101
f(1)=0110
f(2)=01010
f(3)=...
==> f(0) is included in f(2)
```

Cet affichage est celui du programme autonome obtenu en combinant le code extrait avec une petite interface manuelle permettant une saisie simple du début de la suite de mots à tester (voir le contenu de la contribution `Rocq/Higman`). Cette interface permet aussi de tirer des mots au hasard pour tester le comportement de l'algorithme lorsqu'il doit explorer des suites plus longues et/ou des mots plus grands. Mais ces tests se sont rapidement heurtés à un comportement inefficace du programme. En effet, dès que le préfixe de la suite à manipuler contient plus d'une dizaine d'éléments, il devient fréquent d'obtenir une réponse rapide mais décevante :

```
./higman --random
```

.../...

```
.../...
Fatal error: exception Stack_overflow
```

Nous avons également testé une extraction vers Haskell [4]. Comme l'extraction ne génère pas encore de `unsafeCoerce` (fonction équivalente aux `Obj.magic` de Ocaml), nous en avons inséré un manuellement dans la fonction `snake`. On obtient alors un programme ne produisant pas de débordements de pile sur les petits exemples où le programme Ocaml échoue. Mais les temps de calcul sont clairement mauvais, on dépasse par exemple la minute de calcul sur une machine récente pour une suite dont le préfixe à considérer fait moins de dix éléments. Et ces temps de calculs croissent très rapidement lorsque la longueur des préfixes augmente, de façon apparemment exponentielle.

Nos expériences rejoignent donc les analyses antérieures signalant l'efficacité désastreuse de l'algorithme obtenu par A-traduction. De plus, compte tenu de la complexité du code produit, il semble difficile d'analyser a posteriori ce code pour comprendre comment l'améliorer.

5.4.3 La nouvelle preuve de Higman

Nous allons maintenant évoquer une nouvelle démonstration du lemme de Higman en Coq, toujours dans la version restreinte à un alphabet de deux lettres. Cette nouvelle version, datant de 2003, est due à S. Berghofer, et a donné la contribution nommée `Muenchen/Higman`.

Cette section est destinée à illustrer les avancées réalisées depuis 1993 dans la recherche d'une preuve constructive efficace du lemme de Higman. Par contre, contrairement à tous les autres exemples détaillés à partir de ce chapitre, l'extraction de cette nouvelle formalisation ne présente pas d'intérêt du strict point de vue de l'extraction Coq. En effet, les types manipulés ici vont être très simples, sans recours à l'imprédicativité ou à des quantifications d'ordre supérieur. Au final, on obtient du code extrait parfaitement typable en ML, exempt de `Obj.magic`, et que l'ancienne extraction aurait pu gérer.

Une nouvelle approche dans la démonstration

Comme mentionnée dans notre introduction au lemme de Higman, le concept de cette nouvelle formalisation est dû à T. Coquand et D. Fridlender [21]. Il s'agit d'une utilisation d'un principe nommé « bar induction », qui est une forme d'induction sur des arbres à branchement infini. Ce principe de récurrence permet alors de rendre positif le raisonnement par l'absurde de Nash-Williams. En fait, la présente implantation Coq n'est qu'une version adaptée de l'implantation en Isabelle par S. Berghofer. D'autre part une preuve similaire a également été implantée en Alf par D. Fridlender et en Minlog par M. Seisenberger [74]. Nous n'allons pas décrire ici en détail le cheminement de la preuve, fort bien décrite dans [14], p. 104. Signalons simplement qu'au lieu de reposer sur un hypothétique plus petit contre-exemple `Mincex`, on définit un prédicat `bar` sur les suites finies de mots :

[4] Contrairement à l'extraction vers Ocaml, les fichiers correspondant à cette expérimentation ne font pas encore partie de `Rocq/Higman`. Le lecteur intéressé pourra prendre contact avec l'auteur.

```
Inductive L (v : word) : list word → Set :=
  | L0 : ∀w ws, w ⊴ v → L v (w::ws)
  | L1 : ∀w ws, L v ws → L v (w::ws).

Inductive good : list word → Set :=
  | good0 : ∀ws w, L w ws → good (w::ws)
  | good1 : ∀ws w, good ws → good (w::ws).

Inductive bar : list word → Set :=
  | bar1 : ∀ws, good ws → bar ws
  | bar2 : ∀ws, (∀w,bar (w::ws)) → bar ws.
```

Ce prédicat `bar` exprime donc soit que l'on a déjà trouvé nos deux mots imbriqués $w \trianglelefteq v$ dans le préfixe fini courant, soit que tout prolongement de ce préfixe mène plus tard à la vérification du lemme. Il s'agit en fait d'une formulation constructive de la bonne fondation de $\trianglelefteq$. Le point clé de la démonstration est alors d'établir `(bar nil)`. Pour cela on utilise deux autres prédicats inductifs :

```
Inductive R (a : letter) : list word → list word → Set :=
  | R0 : R a nil nil
  | R1 : ∀vs ws w, R a vs ws → R a (w::vs) ((a::w)::ws).

Inductive T (a : letter) : list word → list word → Set :=
  | T0 : ∀b w ws zs, a ≠ b → R b ws zs → T a (w::zs) ((a::w)::zs)
  | T1 : ∀w ws zs, T a ws zs → T a (w::ws) ((a::w)::zs)
  | T2 : ∀b w ws zs, a ≠ b → T a ws zs → T a ws ((b::w)::zs).
```

Et avec ces cinq prédicats (plus $\trianglelefteq$), nous avons fait le tour des structures utilisées par cette formalisation ! Le reste n'est qu'une succession de petits lemmes visant à établir dans un premier temps `(bar nil)`, puis à en extraire les deux indices `i` et `j` recherchés. Au total on obtient une preuve de 220 lignes de Coq, ce qui est remarquablement court[5]. en comparaison des 1085 lignes de la preuve obtenue par A-traduction de la preuve de Nash-Williams.

L'extraction

À l'extraction, on obtient 165 lignes de code, ce qui fait un ratio (script Coq)/(code extrait) extrêmement faible, ratio qui est habituellement plus proche de 10 (voir par exemple `Orsay/FSets` au chapitre 7). D'autre part, ce code extrait est extrêmement simple. Voici par exemple la fonction extraite la plus grande et la plus complexe :

```
let rec prop2 a b xs h ys x zs x0 x1 =
```

…/…

[5]Il est vrai que nous avons retravaillé les preuves de S. Berghofer afin d'utiliser au maximum les possibilités des tactiques automatiques de Coq, ce qui réduit sensiblement la taille de cette contribution, et qui n'a pas été fait dans le cas de `Rocq/Higman`.

.../...

```
match h with
  | Bar1 (ws, g) → Bar1 (zs, (lemma3 ws zs a x0 g))
  | Bar2 (ws, b0) →
      let rec f l0 b1 zs0 h2 h3 =
        match b1 with
          | Bar1 (ws0, g) → Bar1 (zs0, (lemma3 ws0 zs0 b h3 g))
          | Bar2 (ws0, b2) → Bar2 (zs0, (fun w →
              match w with
                | Nil → prop1 zs0
                | Cons (l1, l2) →
                    (match letter_eq_dec l1 a with
                       | Left →
                           prop2 a b (Cons (l2, ws))
                             (b0 l2) ws0 (Bar2 (ws0, b2)) (Cons ((Cons
                             (a, l2)), zs0)) (T1 (l2, ws, zs0, h2)) (T2
                             (a, l2, ws0, zs0, h3))
                       | Right →
                           f (Cons (l2, ws0)) (b2 l2) (Cons ((Cons (b,
                             l2)), zs0)) (T2 (b, l2, ws, zs0, h2)) (T1
                             (l2, ws0, zs0, h3)))))
      in f ys x zs x0 x1
```

Même si les deux branches les plus internes ne sont pas très lisibles à cause d'un affichage imparfait, on reconnaît au moins la structure algorithmique globale de cette fonction, ce qui était très difficile avec `Rocq/Higman`.

Mais le progrès majeur vis-à-vis de `Rocq/Higman` est le gain en vitesse d'exécution. Le programme obtenu peut maintenant manipuler des préfixes de plus d'un millier de mots en quelques dizaines de secondes, chaque mot ayant une longueur aléatoire comprise entre 20 et 80.

Chapitre 6

Réels constructifs et extraction

Dans ce chapitre, nous allons tenter de dresser un bilan des travaux que nous avons menés en collaboration avec plusieurs autres chercheurs européens. L'objectif de ces travaux est l'obtention d'une bibliothèque certifiée d'arithmétique réelle exacte, et ce via l'extraction de formalisations constructives d'analyse réelle en Coq.

Le premier de ces travaux concerne le projet C-CoRN (anciennement FTA) mené à l'université de Nimègue (Nijmegen) aux Pays-Bas. L'étude de l'extraction de certains pans de ce projet a été faite avec deux membres de l'équipe de Nimègue, à savoir L. Cruz-Filipe et B. Spitters. Cette étude a donné lieu à une publication de leur part, dont la référence est [25]. Nous verrons dans la première partie de ce chapitre que beaucoup de chemin a été parcouru depuis que l'idée d'extraire FTA a été lancée, mais également que beaucoup de difficultés subsistent, en particulier concernant l'efficacité du code extrait. Cette première partie de chapitre est très proche du chapitre 6 de [24], puisqu'il s'agit de deux comptes-rendus des mêmes travaux, abordés simplement sous des angles légèrement différents.

Plus récemment, nous avons eu la chance d'avoir de longues discussions avec H. Schwichtenberg concernant l'extraction de nombres réels constructifs lors de l'école d'été 2003 de Marktoberdorf. Et ces discussions ont été prolongées par une visite d'une semaine à Munich effectuée en Septembre 2003. Nous avons alors réalisé ensemble l'ébauche d'une formalisation de réels constructifs en Coq. Cette mini-formalisation est évidemment sans commune mesure avec C-CoRN au point de vue de la taille, mais elle présente l'intérêt d'être dès l'origine pensée en terme d'extraction. Et cette étude a apporté de nouveaux éclairages aux difficultés rencontrées avec l'extraction de C-CoRN.

6.1 L'extraction du projet C-CoRN

6.1.1 Description du projet FTA/ C-CoRN

FTA est l'abréviation anglaise du Théorème Fondamental de l'Algèbre. Ce théorème stipule que tout polynôme non-constant à coefficients complexes possède au moins une racine dans $\mathbb{C}$. En 1999 et 2000, le groupe dirigé par H. Barendregt à l'université de Nimègue a formalisé en Coq une preuve constructive, due à Kneser, de ce résultat [35]. Cette formalisation est réellement impressionnante, la version finale de FTA étant composée de 40 000

lignes de scripts Coq (voir `http://www.cs.kun.nl/gi/projects/fta/`).

Les structures algébriques de FTA

La principale raison d'une telle ampleur est la construction à partir de rien de toute une pile de structures algébriques constructives, la bibliothèque standard de Coq ne comportant pas de structures algébriques adaptées au besoin de cette formalisation. Ces structures sont constituées entre autres de :

- `CSetoids`
- `CSemiGroups`
- `CMonoids`
- `CGroups`
- `CRings`
- `CPolynomials`
- `CFields`
- `COrdFields`
- `CReals`
- `CComplex`

Il est à noter que les `CSetoids` sont basés sur une relation calculatoire de différence, ou « apartness », plutôt que sur une relation d'égalité, qui ne pourrait pas être constructive pour des domaines comme les nombres réels.

D'autre part, ces structures sont imbriquées les unes dans les autres à l'aide de coercions Coq. Par exemple, voici la définition des semi-groupes :

```
Definition is_CSemi_grp (A:CSetoid)(unit:A)(op:CSetoid_bin_op A) :=
  Associative op.

Record CSemi_grp : Type :=
  { csg_crr   :> CSetoid;
    csg_unit  :  csg_crr;                      (* non-empty *)
    csg_op    :  CSetoid_bin_op csg_crr;
    csg_proof :  is_CSemi_grp csg_crr csg_unit csg_op
  }.
```

Un `CSemi_grp` est donc composé d'un `CSetoid` qui est d'une part non-vide (car au moins muni d'un élément), et qui est d'autre part muni d'une opération associative. Sans entrer plus dans les détails, on constate juste que la coercion `:>` assure que tout `CSemi_grp` est également visible comme un `CSetoid`. Il en est de même pour les autres structures algébriques : chacune s'appuie sur une structure précédente, et y rajoute des objets et/ou des propriétés. On obtient ainsi une chaîne linéaire de coercions allant d'un `CComplex` à un `CSetoid`, les polynômes formant la seule structure qui s'écarte de cette chaîne continue.

La définition de ces structures et leurs propriétés de base occupent entre le tiers et la moitié des 40 000 lignes évoquées précédemment, dont une bonne partie consacrée aux propriétés

de base des polynômes. En fait, une fois prouvés le théorème des valeurs intermédiaires[1] et l'existence de racines n-ièmes[2] dans $\mathbb{R}$ et $\mathbb{C}$, il ne reste plus qu'environ 3 000 lignes consacrées au lemme de Kneser et à FTA proprement dit[3].

Des réels axiomatisés, puis réalisés

Autre point crucial de l'architecture de FTA, il faut noter que les nombres réels ont été utilisés de manière « axiomatique » [36]. En effet, une fois défini dans `CReals.v` le type Coq des structures algébriques isomorphes aux nombres réels, tout le reste de la preuve de FTA se fait par rapport à une telle structure particulière. En fait, pour des raisons de convenance technique[4], cette structure est posée en axiome :

```
Axiom IR : CReals.
```

De la sorte, toute représentation particulière des nombres réels constructifs peut remplacer cet axiome `IR`. Et une telle représentation particulière `Concrete_R` a été développée par M. Niqui a posteriori, basée sur les suites de Cauchy. La transformation de l'axiome précédent en la définition suivante ne change alors normalement pas le caractère valide du reste de FTA :

```
Definition IR : CReals := Concrete_R.
```

De FTA à C-CoRN

Depuis l'achèvement de la preuve de FTA proprement dite, cette formalisation a été progressivement réorganisée en un projet plus ambitieux. La preuve du théorème fondamental de l'algèbre n'est plus maintenant qu'une des facettes de ce nouveau projet C-CoRN (pour Constructive Coq Repository at Nijmegen). Celui-ci a vocation à devenir une bibliothèque de résultats mathématiques constructifs basés sur la hiérarchie des structures algébriques décrites précédemment. C-CoRN regroupe déjà, outre FTA, des résultats étendus concernant les séries, les fonctions transcendantes usuelles, et surtout une partie nommée FTC (pour Fondamental Theorem of Calculus). L. Cruz-Filipe y établit qu'intégration et dérivation sont deux processus réciproques [23].

6.1.2 Les premières tentatives d'extraction

Notre premier contact avec FTA date d'octobre 2001. Dans un mail, M. Niqui relatait ainsi sa première tentative d'extraction de FTA : « *after 24 hours my machine ran out of memory* », la machine en question étant pourtant plus que raisonnable pour l'époque.

Il faut bien dire que FTA n'a pas été à l'origine conçu en vue d'une extraction. Bien sûr, comme ce développement est constructif, ses auteurs étaient conscients de la possibilité

[1] Voir le fichier `IVT.v`.
[2] Voir les fichiers `NRootIR.v` et `NRootCC.v`.
[3] Voir les fichiers `KeyLemma.v`, `MainLemma.v`, `KneserLemma.v`, `FTAreg.v` et `FTA.v`.
[4] Les modules et foncteurs n'existaient pas alors en Coq.

théorique d'en extraire un programme. Mais cette possibilité, jugée irréaliste à l'époque, n'a pas influencé les choix de conception initiaux. La preuve en est que tout le développement a été placé à l'origine dans la sorte `Prop` des objets ignorés par l'extraction. Pour ses premières tentatives d'extraction, M. Niqui a alors remplacé simplement toutes les occurrences de `Prop` par `Set`. Au lieu de tout ignorer, l'extraction se met alors à tout conserver et tout traduire. Comme ce développement de 40 000 lignes de script Coq génère des termes de preuves complexes et volumineux, on comprend alors que son extraction est excessivement plus gourmande que celle des autres exemples rencontrés jusque-là, d'où les problèmes de temps de calcul et d'occupation mémoire de M. Niqui.

En fait, il est apparu par la suite que l'extraction de cette version de FTA utilisant `Set` comme univers n'était pas si infaisable que cela, à condition tout d'abord d'utiliser du matériel beaucoup plus récent, et d'autre part de corriger l'extraction en supprimant une « optimisation » issue d'une heuristique plus qu'hasardeuse, qui avait pour conséquence le remplacement de trop de constantes par leurs définitions. Dans le cas particulier de FTA, cela conduisait en effet à une augmentation faramineuse de la taille du code. Voici ce que signalait L. Cruz-Filipe en février 2003 :

> *You might find it interesting to know that we actually managed to extract the *original* FTA version (e.g. the one currently in the Coq library, with all the logic in Set) after you fixed the inlining bug in the extraction routine. We still almost ran out of resources (on a 2GHz machine with 1Gb RAM memory and 2Gb swap), but the extracted code is "only" around 13 Mb.*

Ce chiffre de 13 Mo de code source semble réellement aberrant pour un programme qui, rappelons-le, doit juste calculer des approximations de racines de polynômes complexes. On peut néanmoins relativiser quelque peu ces 13 Mo. Ils sont en effet dus en grande partie à deux phénomènes désagréables, mais simples :

- Le premier problème tient aux tentatives faites par l'extraction d'embellir l'affichage (« pretty-print » en anglais). Cet affichage se fait au moyen de boites d'affichages, verticales ou horizontales, fournies par le langage Ocaml. Or ce mécanisme fonctionne avec une largeur de ligne fixée, qui est de 80 colonnes par défaut. Dans le cas des fonctions démesurées de FTA, respecter à la fois l'indentation et cette limite de largeur conduit fréquemment à n'utiliser que le dernier quart des lignes, voire moins encore. Au final, le fichier extrait est constitué pour plus de la moitié de caractères blancs en début de lignes.

- Lors d'un survol de cet énorme fichier extrait, l'autre point qui saute aux yeux est l'omniprésence des projections issues de coercions. Par exemple, sur une structure comme celle des nombres réels, l'addition se fait en considérant $\mathbb{R}$ comme un semi-groupe `CSemi_grp`, grâce à une succession de coercions, puis en utilisant l'opérateur présent dans le champ `csg_op` de cette structure de `CSemi_grp`. En fait, l'utilisateur de Coq n'a pas à se soucier de ces coercions, qui sont implicites. Si `IR` est la structure des réels, on peut former directement `IR.csg_op`. Mais ces coercions sont stockées explicitement dans le terme de preuve, et se retrouvent donc également dans le terme extrait final. Ainsi l'extraction de l'addition réelle est :

```
iR.crl_crr.cof_crr.cf_crr.cr_crr.cg_crr.cm_crr.csg_op
```

On reconnaît là les projections successives vers les sous-structures de corps ordonné, puis de corps, puis d'anneau, puis de groupe, puis de monoïde, et enfin de semi-groupe, avant la projection finale vers le champ contenant l'opérateur d'addition. Au final, il faut quasiment une ligne pour écrire cette addition, qui apparaît plusieurs centaines de fois dans le fichier extrait. Et il en va de même pour les autres objets élémentaires comme `0`, `1` et les opérations restantes.

Il est bien sûr envisageable de factoriser le code produit pour éviter, au moins en grande partie, de telles chaînes de projections. Il suffirait ainsi de définir une fois pour toute une constante `iR_plus` égale à l'objet précédent. Mais autant cela sera facilement réalisable pour une structure fixée comme `iR`, autant il serait beaucoup plus compliqué et moins efficace de le faire dans la première moitié du fichier extrait, où l'on raisonne sur des structures inconnues, passées en argument des fonctions.

En fait, même en factorisant manuellement ces espaces et ces chaînes de coercions, on obtient encore un code source de plusieurs mégaoctets. Il semble donc évident qu'une partie substantielle de ce code n'est en fait d'aucun intérêt au niveau algorithmique.

6.1.3 Distinction entre parties logiques et parties calculatoires

Pour permettre à l'extraction d'éliminer au moins partiellement le code mort de ces termes extraits, L. Cruz-Filipe et B. Spitters ont alors repris le développement initial, en tentant d'identifier les parties pouvant rester dans `Prop`. Ce travail est décrit en détail dans [25] et dans le début du chapitre 6 de [24].

Il faut bien comprendre que FTA est un développement atypique quant à la distinction entre parties calculatoires et parties logiques. Usuellement, il est assez facile d'opérer la distinction entre des opérations calculatoires comme `1+2` et des assertions logiques comme `n=0∨0<n`. Mais dans FTA, il est crucial que la relation `<` sur les réels soit calculatoire : derrière `x<y` se cache en fait un rationnel strictement positif minorant `y-x`. Et ce rationnel va effectivement être utilisé en pratique dans des opérations, comme par exemple pour calculer l'inverse d'un réel non nul (voir par exemple p.148 de [24]). De même, la différence entre deux réels a un contenu calculatoire.

À l'opposé, l'égalité et la relation $\leq$ ne sont pas décidables sur les réels et peuvent donc rester logiques. En fait, ces deux relations sont définies comme les négations respectives des relations de différence et d'ordre strict `<`. Cela leur confère bien un statut logique, car `False`, utilisé dans les négations, reste bien toujours dans `Prop`.

Ces décisions sur les relations de base influent ensuite sur le statut d'opérateurs comme les disjonctions ou les conjonctions. Par exemple, on ne pourra pas utiliser la disjonction logique `or` usuelle dans l'exemple `n=0∨0<n`, car sa partie droite est informative. Et en particulier cette expression n'est pas équivalente constructivement à `0≤n`, qui est complètement logique.

On voit donc qu'opérer la distinction `Prop/Set` dans le cas de FTA n'est pas trivial, ce qui explique le choix du « tout logique » initial, puis le passage au « tout informatique »

ensuite. L. Cruz-Filipe et B. Spitters ont en particulier dû remanier certaines portions du développement initial, à la manière du `or` précédent remplacé par une disjonction ad hoc. Mais cela en valait la peine, puisqu'on obtient un gain de l'ordre d'un facteur 10 sur la taille du code extrait.

Outre cette répartition entre `Prop` et `Set` des constructions de FTA, L. Cruz-Filipe et B. Spitters ont également constaté qu'il était possible de faire encore chuter la taille des termes extraits grâce à quelques modifications ingénieuses des preuves portant sur $<$ et $\leq$. Tout d'abord, ils ont exploité le fait que les deux formulations suivantes de la propriété de Cauchy sont équivalentes :

$$\forall \varepsilon > 0 \;\; \exists N : \texttt{nat} \;\; \forall m, n > N \;\; |x_m - x_n| < \varepsilon$$

et

$$\forall \varepsilon > 0 \;\; \exists N : \texttt{nat} \;\; \forall m, n > N \;\; |x_m - x_n| \leq \varepsilon$$

Utiliser la seconde variante conduit alors à faire disparaître des termes extraits la majeure partie de ces propriétés de Cauchy, pour ne garder que la partie essentielle, à savoir celle construisant effectivement la borne N à partir d'un ε donné.

Un autre exemple d'optimisation concerne les preuves de propriétés de la forme $a < b$. À l'origine, ces preuves se faisaient fréquemment par des raisonnements successifs du genre $a < x_1 < x_2 < x3 < b$. En fait, une seule de ces étapes nécessite réellement un ordre strict : il suffit par exemple d'établir $a < x_1 \leq x_2 \leq x_3 \leq b$, et seule cette première étape de raisonnement subsistera à l'extraction.

Au final, grâce à de telles transformations des preuves de FTA, ainsi qu'à la refonte de la division, L. Cruz-Filipe et B. Spitters ont fait chuter la taille du code extrait à un peu plus de 200 Ko, soit presque 100 fois moins que lors des premières extractions réussies. De plus, cette extraction s'effectue désormais en quelques secondes, alors même que notre implantation de l'outil d'extraction n'a jamais eu la vitesse d'extraction comme critère de conception. Enfin, cette taille de 200 Ko comprend toujours de multiples redondances liées aux projections des coercions, qui n'ont pas encore été résolues de façon satisfaisante. Par contre, la diminution de taille du code extrait a eu pour conséquence de grandement atténuer les problèmes d'indentation du code extrait.

6.1.4 Compilation du code extrait

Une fois le code extrait de FTA ramené à l'échelle plus raisonnable de quelques milliers de lignes, le défi suivant consiste à tenter d'en faire un programme qui puisse s'exécuter et retourner véritablement un résultat utile. En fait, lors des premières extractions réussies de FTA, on butait immédiatement sur un obstacle de taille, à savoir la non-typabilité de ce code extrait. Et on est bien loin de la situation des contributions `Lannion` et `Higman` du chapitre précédent, pour lesquels il était encore envisageable d'insérer soi-même un ou deux `om` à la main. Ici, les conflits de typage se chiffrent en centaines, ce qui rend le recours à une méthode automatique quasiment indispensable. Heureusement, nous avons depuis lors mis au point l'insertion automatique de `Obj.magic` à l'emplacement des conflits de typage (voir le chapitre 3). Dans le cas présent, environ 400 de ces `Obj.magic` sont insérés dans

le code extrait. Et celui-ci est effectivement accepté par le compilateur Ocaml sans aucune manipulation supplémentaire.

Si l'on regarde de plus près ces conflits de typage, on retrouve quelques-unes des situations évoquées au chapitre 3. Par exemple, le premier problème qui apparaît concerne la structure de `CSetoid` :

```
Record CSetoid : Type :=
  { cs_crr   :> Set;
    cs_eq    :  (Relation cs_crr);
    cs_ap    :  (Crelation cs_crr);
    cs_proof :  (is_CSetoid cs_crr cs_eq cs_ap)
  }.
```

Un `CSetoid` est donc un type quelconque muni de relations d'égalité et de différence. En fait, il s'agit là d'un exemple similaire à l'inductif `any` de la page 92. Dans les deux cas, on embarque un type dans une structure, et l'on fait dépendre de ce type interne les champs ultérieurs. Cette structure n'a pas de contrepartie[5] en ML, et l'extraction produit l'approximation suivante, dans laquelle `cs_crr` a été remplacé par le type inconnu $\mathbb{T}$ (ou `__`) :

```
type cSetoid = { cs_ap : __ crelation; cs_proof : (__, __) is_CSetoid }
```

Et l'utilisation des champs `cs_ap` ou `cs_proof` conduit alors fréquemment à des situations non-typables, qui forcent le recours à des `Obj.magic`.

6.1.5 L'exécution du programme extrait

Une fois réglé la question de la compilation, nous avons alors réalisé une petite interface d'entrée-sortie afin de pouvoir utiliser ce code et tester son efficacité. Comme FTA manipule les nombres réels sous forme de suites de Cauchy, notre interface se contente pour quelques réels x fixés de demander un rang n, et de retourner le nombre rationnel x_n constituant le n-ième élément de la suite de Cauchy x. Plus exactement, cette interface *tente* de retourner x_n. Il est en effet très vite apparu que ce programme extrait souffre d'énormes problèmes d'efficacité. En particulier, dans le cas de l'exemple « canonique » que constitue le calcul de $\sqrt{2}$ vu comme racine du polynôme $x^2 - 2$, le programme semble parti pour tourner quelques siècles, sauf dans le cas de la première approximation ... qui vaut zéro. Comme le dit L. Cruz-Filipe, « la seule bonne nouvelle était que ce programme ne nécessitait que très peu de mémoire pour s'exécuter ».

En fait, rendre efficace ce programme extrait est un défi au moins aussi ardu que celui consistant à donner une taille raisonnable au code extrait. Et la résolution de ce nouveau défi est encore en cours aujourd'hui, même si beaucoup de chemin a déjà été parcouru. La suite de cette section relate les différentes transformations et optimisations que nous avons fait subir à FTA, et les progrès ainsi réalisés. Il est à noter que cette partie correspond à une phase

[5]En fait, Ocaml 3.07 permet dans les enregistrements une certaine forme d'abstraction sur les types. On peut ainsi écrire `type setoid = {eq : 'a.'a→'a→bool}`. Mais cela ne convient pas réellement à nos besoins ici, car une égalité particulière `eq_int : int→int→bool` ne peut servir à bâtir un tel `setoid`.

de collaboration très active avec l'équipe de Nimègue et en particulier avec L. Cruz-Filipe, alors que notre contribution personnelle à cette étude de FTA s'était essentiellement limitée jusque-là à corriger le dysfonctionnement de l'extraction conduisant à trop de dépliages de constantes.

Amélioration des types de données

Notre premier réflexe pour améliorer le programme extrait a été de faire la chasse aux types de données inefficaces. En effet, cette seule étape avait permis auparavant de produire des programmes raisonnables à partir de développements non initialement prévus pour l'extraction, comme par exemple la contribution `Bordeaux/Additions` de P. Castéran. Dans le cas de FTA, une première modification a concerné les nombres rationnels utilisés par M. Niqui pour modéliser les nombres réels via des suites de Cauchy. Ces nombres rationnels sont en effet à l'origine des enregistrements où le premier champ est un entier relatif représentant le numérateur de la fraction, et le second champ est un entier naturel codant le dénominateur. Mais autant les entiers relatifs sont efficaces car codés de façon binaire dans le type `Z` de Coq, autant les entiers naturels `nat` choisis à l'origine sont codés de façon unaire. Notre première contribution à FTA a donc été de remplacer dans les dénominateurs le type `nat` par le type `positive` des entiers strictement positifs, encodés sous forme binaire. Non seulement cela a grandement accéléré les opérations sur les nombres rationnels, mais cela a même simplifié la formalisation : il n'y a plus désormais besoin de se préoccuper du cas d'un dénominateur nul.

Une refonte du modèle des nombres réels

En fait, la modification précédente a été assez décevante. Bien sûr, cette amélioration des nombres rationnels méritait d'être faite, mais elle n'a pas induit d'effet visible sur les tests d'alors : les calculs qui divergeaient continuaient à le faire. Pire, le code extrait a persisté, malgré ce changement, à contenir d'innombrables valeurs dans `nat`, ces valeurs allant ainsi jusqu'à la constante 48. En fait, ce genre de constantes servait essentiellement lors de la construction d'un modèle des réels par le biais de suites de Cauchy. Et L. Cruz-Filipe a finalement compris comment remanier complètement cette partie, de sorte qu'une preuve gigantesque comme `Rmult_is_extensional`[6], longue de 1800 lignes de script Coq, s'obtient maintenant en une dizaine de lignes. Et la taille du code extrait décroît encore, la plupart de ces constantes volumineuses de type `nat` disparaissent ... mais pourtant l'efficacité n'est toujours pas au rendez-vous concernant le calcul de $\sqrt{2}$ via FTA.

Des tests moins ambitieux

Devant de telles difficultés, nous avons alors tenté de mieux cerner les causes des inefficacités via des tests plus progressifs. Il est alors apparu que tous les calculs d'arithmétique

[6]Ce résultat stipule que lorsque l'on a des réels tels que `a*b≠a'*b'`, alors `a≠a'` ou bien `b≠b'`. Il s'agit du dual de l'énoncé usuel sur l'égalité, qui affirme que deux multiplications de nombres égaux produisent des nombres égaux.

sur des nombres rationnels sont quasiment instantanés, même lorsque ces rationnels sont vus comme des réels[7] ou encore lorsque ces calculs se font via l'intermédiaire de polynômes. A l'opposé, le calcul de $\sqrt{2}$ comme réciproque de 2 par la fonction $\lambda x.x^2$ diverge également, alors même que cette façon d'obtenir $\sqrt{2}$ est normalement beaucoup plus simple que l'utilisation de FTA sur le polynôme $x^2 - 2$. En cherchant des exemples de nombres réels simples mais ayant néanmoins des suites de Cauchy non constantes, L. Cruz-Filipe a suggéré de s'intéresser aux limites de séries, et en particulier à $e = \sum_{n=0}^{\infty} \frac{1}{n!}$ et dans une moindre mesure à $\pi = 4\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$. Pour la première fois, nous avons alors obtenu des calculs non-triviaux qui terminaient. Voici par exemple les fractions retournées comme premiers termes de la suite de Cauchy représentant la constante e. Sans surprise, il s'agit des sommes partielles de la série qui précède.

rang	fraction	valeur
0	0/1	0.000000000
1	1/1	1.000000000
2	2/1	**2**.000000000
3	5/2	**2**.500000000
4	32/12	**2**.666666666
5	780/288	**2.7**08333333
6	93888/34560	**2.71**6666666
7	67633920/24883200	**2.718**055555
8	340899840000/125411328000	**2.7182**53968
9	13745206960128000/5056584744960000	**2.7182**78769
10	4987865758275993600000/1834933472251084800000	**2.71828**1525
11	1809996909856539782676480000/665860658410473652224000000	**2.71828**101

En fait, nous n'avons pas pu obtenir de telles fractions lors des premiers tests, puisque calculer l'approximation de rang 7 prenait déjà plus d'une heure, pour expérimentalement ne donner que trois chiffres corrects. La raison d'une telle efficacité limitée réside dans la manière de calculer les termes $\frac{1}{\mathtt{n!}}$ de la série. En effet à l'origine, le nombre réel `n!` était d'abord calculé dans le type `nat` via la factorielle `fac : nat→nat`, puis seulement ensuite injecté vers les réels via la fonction `nring : ∀R:CRing, nat→R` plus quelques coercions. Et cette dernière fonction se contente de transformer un entier codé en unaire en une suite d'additions $1 + \ldots + 1$ dans l'anneau `R` considéré, ici celui des réels. Cette manière de faire impliquait donc de très nombreux calculs sur des factoriels avec un codage *unaire* des nombres. Nous avons alors proposé de passer le calcul des factoriels dans le type `positive`, suivi de l'usage d'une fonction `pring : ∀R:CRing, positive→R`, qui injecte des entiers vers un anneau en tirant parti de leurs codages binaires.

Malheureusement, cette amélioration n'a pas eu l'effet escompté. Après investigation, il est apparu que le facteur limitant suivant était l'inversion de `n!` en $\frac{1}{\mathtt{n!}}$. En fait, dans FTA, la division n'est pas un opérateur binaire `a/b` mais un opérateur ternaire `a/b//h`, où `h` est une preuve de non-nullité de `b`. Et comme nous l'avons déjà mentionné précédemment, le

[7] On considère alors des suites de Cauchy constantes.

contenu calculatoire de `h` sert effectivement à la construction de la division. Dans le cas qui nous occupe, la preuve de non-nullité de `n!` était donnée par un terme `(fac_ap_zero n)`, et à l'origine, la structure de ce terme le rendait isomorphe à `n!` codé en unaire. Il s'agissait en effet d'une preuve de la forme $0 < 1 < \ldots < \mathtt{n!}$. L'utilisation de techniques décrites précédemment permet alors de passer à une preuve de la forme $0 < 1 \leq \ldots \leq \mathtt{n!}$. La nouvelle complexité estimée du calcul de $\frac{1}{\mathtt{n!}}$ est alors de l'ordre de la taille de l'écriture en base deux de `n!`, soit $\ln(\mathtt{n!}) \sim_\infty \mathtt{n}\ln(\mathtt{n})$. Et effectivement, cette version permet d'obtenir en quelques heures les fractions présentées précédemment, ce qui était impossible auparavant.

Expérimentalement, on constate que le calcul de l'approximation de rang $k+1$ demande environ dix fois plus de temps que le calcul pour le rang k, et qu'en même temps la précision augmente également d'un facteur dix : en moyenne, on obtient un nouveau chiffre décimal correct à chaque étape. On voit donc que l'extraction est capable de manipuler des fractions de taille conséquente, mais que la rapidité laisse encore à désirer : à ce rythme, il faudrait un calcul de plusieurs mois pour obtenir dix décimales correctes.

Le problème de la duplication des calculs

Pour comprendre plus finement le déroulement des calculs, nous avons alors utilisé des techniques de « profilage » des exécutions. Et nous avons constaté des choses curieuses. Par exemple, le calcul de l'approximation de rang 7 engendrait 14 passages dans une fonction nommée `e_series`, qui calcule pour un `n` donné la valeur de $\frac{1}{\mathtt{n!}}$. Comme l'approximation demandée ne devait utiliser que les 7 premiers termes de la série, nous en avons déduit que les calculs de ces termes étaient dupliqués. Initialement, nous avons pensé que cette redondance des calculs venait de l'usage de `e_series` dans la fonction `e_series_conv : (convergent e_series)`, qui prouve la convergence de la série dont les termes sont donnés par `e_series`. Mais il s'agit d'une fausse piste, car `e_series_conv` n'apparaît que dans un champ de structure que l'on n'utilise jamais. En fait, nous avons fini par remonter à l'origine de la duplication, située dans la définition suivante :

```
Definition LimR_CauchySeq (a:CauchySeq R_COrdField') :=
  Build_CauchySeq F
    (fun m ⇒ CS_seq F (CS_seq _ a m) (T (CS_seq _ a m) m))
    (CS_seq_diagonal a).
```

Le type `R_COrdField'` est l'ensemble des suites de Cauchy bâties sur un corps ordonné archimédien. Et la construction `LimR_CauchySeq` permet d'exhiber la limite d'une suite de Cauchy de telles suites, au moyen d'un argument diagonal. Le problème ici est la répétition de `(CS_seq _ a m)`. Si l'on veut éviter un double calcul lorsque la suite `a` est instanciée par `e_series`, il faut effectuer une factorisation :

```
Definition LimR_CauchySeq (a:CauchySeq R_COrdField') :=
  Build_CauchySeq F
    (fun m ⇒ let b := CS_seq a m in CS_seq F b (T b m))
```

…/…

```
...
    (CS_seq_diagonal a).
```

Ici, les deux occurrences de ce sous-terme dupliqué se trouvaient sous un même contexte. On peux donc imaginer un outil automatique capable d'identifier cette redondance et de la factoriser. Mais outre son coût, une telle passe de factorisation devra décider que faire dans des situations plus complexes, par exemple lorsque ces occurrences multiples peuvent ne pas être exécutées du tout. Doit-on alors risquer d'occasionner des calculs supplémentaires en créant un « let-in » ?.

Dans FTA, on rencontre également très fréquemment des duplications liées aux types dépendants. Par exemple, pour construire la fraction $\frac{1}{5}$ dans un corps `F`, on peut utiliser la division ternaire `One/(pring R 5)//(pring_ap_zero F 5)`. Or la partie preuve de non-nullité, `(pring_ap_zero F 5)`, de type `(pring F 5)≠Zero`, contient presque à coup sûr le terme `(pring F 5)`, qui est donc au moins calculé deux fois lors de cette division. De même, notre nouvelle preuve de `fac_ap_zero` de la section précédente contient encore une occurrence de `n!`, même si cette preuve est maintenant de la forme $0 < 1 \leq \ldots \leq$ `n!`.

Cette situation est réellement banale dans FTA, alors que dans un développement plus standard la partie preuve serait purement logique, et disparaîtrait sans occasionner de calculs redondants. Parfois heureusement, la duplication est potentiellement présente dans FTA, mais ignorée, comme c'est le cas pour `e_series` ci-dessus. Mais ces duplications peuvent également changer la complexité du tout au tout, pour peu qu'elles interviennent au sein de fonctions récursives. Il s'agit là d'une des causes probables de l'inefficacité du programme extrait de FTA. Et corriger automatiquement ce type de redondances au niveau de l'extraction serait vraiment délicat, car le problème peut être réparti entre plusieurs fonctions, comme avec la division, `pring` et `pring_ap_zero`. La seule réponse actuelle à ces duplications est une analyse manuelle a posteriori, par `profilage`, ce qui est long et pénible, et ne permet de repérer que les problèmes les plus flagrants.

Une axiomatisation des réels trop contraignante

Quant on y repense, il est vraiment déconcertant de devoir consacrer autant d'efforts afin d'obtenir une solution efficace pour le calcul d'un nombre réel comme $\frac{1}{n!}$, qui n'est après tout qu'un rationnel. Pourquoi ne pas faire ce calcul dans la structure des nombres rationnels, et l'injecter seulement dans un second temps vers les nombres réels ?

Malheureusement pour l'extraction, il n'est pas possible d'opérer ainsi directement. Le problème se situe au niveau de la distinction dans FTA entre la structure abstraite des nombres réels, et sa contrepartie concrète. En fait, toutes les parties impliquant des réels, comme par exemple la preuve de FTA proprement dite, utilise une structure abstraite des réels, axiomatisée :

```
Axiom IR : CReals.
```

Et ce type `CReals` est spécifié comme étant un corps ordonné archimédien où toute suite de Cauchy admet une limite. On a donc seulement à notre disposition un nombre minimal d'objets primitifs et de propriétés de bases, issues des structures sous-jacentes.

Ainsi, les seuls réels primitifs connus sont 0 et 1, qui sont respectivement les éléments

`csg_unit` et `cr_one` des structures de semi-groupe et d'anneau de `IR`. Au lieu d'injecter directement un rationnel dans `IR`, on doit le reconstruire en utilisant 0, 1 et les opérations $+$, $*$ et $/$, cette dernière opération exigeant de plus une preuve de non-nullité en troisième argument. De telles preuves passent en fait souvent par des preuves de stricte positivité ou négativité, que l'on doit également bâtir à partir d'un noyau restreint de propriétés de base. Voici en effet les seules propriétés connues à l'origine concernant l'ordre $<$ sur `IR` :

- l'antisymétrie
- la transitivité
- la compatibilité vis à vis de l'addition : $x < y$ implique $x + z < y + z$
- la conservation de la positivité par multiplication : $0 < x$ et $0 < y$ impliquent $0 < x * y$
- la dichotomie : $x \neq y$ implique ou bien $x < y$, ou bien $y < x$
- la propriété d'Archimède, stipulant tout nombre réel est majoré par l'injection dans `IR` d'un entier adéquat.

Une propriété aussi basique que $0 < 1$ n'est donc pas primitive, mais dérivée des propriétés précédentes. Et il en est évidemment de même pour des preuves plus complexes comme la preuve précédente de $0 <$ `n!`.

A l'opposé, FTA fournit également un modèle concret nommé `Concrete_R` des nombres réels, bâti sur les suites de Cauchy de nombres rationnels. Et dans ce modèle, il est immédiat d'injecter un rationnel `q` dans `Concrete_R` via une fonction `inject_Q : Q→Concrete_R`. Il suffit en effet de prendre la suite de Cauchy dont tous les termes valent `q`. De même, les preuves de la forme `a<b` peuvent être beaucoup plus directes dans `Concrete_R`, puisque l'on a accès à la *définition* de `<`, qui demande l'existence d'un rationnel strictement positif Δ et d'un rang `N` au-delà duquel les termes des deux suites de Cauchy à comparer sont toujours écartés de plus de Δ. En particulier, pour deux rationnels `q<q'`, on obtient immédiatement `(inject_Q q)<(inject_Q q')` dans `Concrete_R`, en prenant `Δ=q'-q` et `N=0`.

Cette séparation entre réels abstraits et réels concrets est sans doute bénéfique au niveau mathématique, car elle assure que les preuves faites au niveau abstrait sont indépendantes de la représentation particulière choisie au niveau concret. On peut ultérieurement changer ce modèle concret sans risque pour la pérennité des preuves abstraites. Par contre, du point de vue de la programmation, on se trouve en présence de deux modules interagissant via une interface trop minimaliste, ce qui oblige le module de haut niveau à réinventer fréquemment la roue, et ce de manière inefficace. Que dirait-on d'un module d'arithmétique entière dont l'interface n'exporterait pas la multiplication, sous prétexte qu'on peut la simuler en répétant des additions ?

Pour confirmer que cette distinction entre les deux niveaux concrets et abstraits constituait bien un goulot d'étranglement pour le programme extrait, nous avons ajouté au niveau abstrait un certain nombres d'axiomes, que nous avons ensuite réalisés au niveau concret. Puis nous avons signalé à l'extraction de remplacer les axiomes par les réalisations concrètes. Cela a eu un effet spectaculaire sur les performances du test calculant des approximations de la constante d'Euler. Au lieu d'obtenir péniblement l'approximation de rang 11 en plus d'une heure, nous pouvons maintenant calculer celle de rang 100 en 77 secondes. La fraction

obtenue remplit deux pages d'écran, et donne 157 décimales correctes. Quant à la complexité, elle semble seulement doubler tous les 10 rangs, au lieu d'être décuplée à chaque rang supplémentaire. Voici quelques détails sur le code permettant cela. Tout d'abord, nous effectuons un certain nombre de définitions au niveau concret :

```
(* Injection directe de la factorielle dans Concrete_R via inject_Q *)
Definition concrete_fact (n:nat) : Concrete_R :=
  inject_Q Q_as_COrdField (inject_Z (pos_fact n)).

Lemma concrete_fact_ap_zero : ∀n:nat,(concrete_fact n)[#]Zero.
 intros; red; simpl; unfold R_ap. (* retour à la définition de ≠ *)
 right; unfold R_lt. (* retour à la définition de < *)
 exists 0. (* le rang N *)
 exists (inject_Z 1). (* un rationnel str. positif séparant 0 et n! *)
 .... (* la suite est dans Prop *)

(* Le lien entre la nouvelle factorielle et l'ancienne *)
Lemma concrete_fact_pos_fact :
 ∀n:nat,(pring Concrete_R (pos_fact n))[=](concrete_fact n).
```

Ensuite, nous ajoutons quelques axiomes au niveau abstrait, celui de `IR` :

```
Axiom concrete_fact' : nat → IR.
Axiom concrete_fact_ap_zero' : ∀n:nat,(concrete_fact' n)[#]Zero.
Axiom concrete_fact_pos_fact' :
  ∀n:nat,(pring IR (pos_fact n))[=](concrete_fact' n).

Definition concrete_e_series :=
  fun n ⇒ One[/]?[//](concrete_fact_ap_zero' n).

Lemma concrete_e_series_conv : convergent concrete_e_series.
...
(* preuve comme avant,en utilisant l'égalité concrete_fact_pos_fact'
   pour se ramener au cas précédent. *)

Definition concrete_E := series_sum ? concrete_e_series_conv.
```

Enfin, il faut signaler à l'extraction ce qu'elle doit faire des axiomes :

```
Extract Constant concrete_fact' ⇒ concrete_fact.
Extract Constant concrete_fact_ap_zero' ⇒ concrete_fact_ap_zero.
```

On notera qu'il n'y a pas besoin de déclarer le troisième axiome, car il est logique.

Il est intéressant de noter au passage que notre formalisation de $\mathbb{Q}$ à l'aide de paires `Z*positive` fonctionne correctement, en tout cas pour nos besoins actuels. Bien sûr, on peut trouver mieux encore, puisque Maple ou Mathematica sont capables de retourner la grande fraction de rang 100 en quelques dixièmes de secondes. Mais ces calculs rationnels ne constituent certainement pas un facteur limitant pour FTA.

Signalons enfin une dernière expérimentation, qui tentait de trouver une troisième voie

entre tout faire dans `IR` et tout faire dans `Concrete_R`. En effet, comme nous l'avons déjà signalé, cette distinction entre `IR` et `Concrete_R` a ses intérêts, et d'autre part tout réécrire dans `Concrete_R` deviendrait vite pénible. Nous avons donc essayé de ne remplacer que certaines preuves critiques de `IR` par un équivalent dans `Concrete_R`. Mais nos tentatives sur `fac_ap_zero` n'ont apporté qu'un gain minime par rapport aux temps initiaux, sans commune mesure avec le gain apporté par `concrete_E`.

6.2 Des réels alternatifs dédiés à l'extraction

Notre dernière tentative d'amélioration de l'extraction de FTA a consisté à analyser le comportement du calcul de $\sqrt{2}$ vu comme réciproque de 2 par la fonction de mise au carré. Mais faute de temps, nous avons seulement pu déterminer que l'inefficacité résidait dans certaines sous-fonctions traitant de polynômes, point sur lequel nous reviendrons. Nous allons maintenant voir comment une petite formalisation de réels constructifs, indépendante de FTA, nous a permis de cerner quelques points critiques expliquant l'efficacité ou l'inefficacité d'un tel calcul de $\sqrt{2}$.

Cette petite formalisation de réels constructifs a été réalisée en collaboration avec H. Schwichtenberg, à la suite de longues discussions que nous avons eu lors de l'école d'été de Marktoberdorf 2003, à propos de son cours d'analyse constructive [73] et d'extraction en Coq. Cette étude a ensuite été prolongée lors d'une visite d'une semaine à Munich en septembre 2003. Sans avoir aucunement vocation à concurrencer FTA/C-CoRN, cette étude vise à considérer immédiatement les réels constructifs du point de vue de l'extraction, à la différence de FTA où cette idée d'extraction est venue a posteriori. Pour justifier cette nouvelle étude, voici tout de suite son principal résultat :

```
1850738521931038153706479986072768566074474889952922673412495088628037078498212257925892008186006084221171975185924353893529607482952 7 / 1308669759060604982435085250362633629384375727808179217478381261103282433564486174203616957499871349117105758599860865929629285891 3867
```

Cette fraction respectable est une approximation de $\sqrt{2}$ avec plus de 140 chiffres binaires corrects, soit 42 décimales justes. Et nous avons obtenu ce résultat en 3 minutes environ, en utilisant le même principe que lors de nos tests infructueux avec FTA, à savoir la recherche d'une valeur réciproque de 2 par la fonction de mise au carré.

Il faut préciser immédiatement que ce petit développement n'établit pas de résultat général comme FTA, mais au contraire est spécialisé au calcul de $\sqrt{2}$. De plus, comme nous allons le voir, certaines parties sont restées inachevées et posées en axiomes. Il ne s'agit donc pas d'une formalisation complète, mais plutôt d'une étude de faisabilité. Ceci étant dit, ces quelques centaines de lignes se sont révélées pleines d'enseignements.

6.2.1 La méthode de développement

L'objectif était donc de pouvoir au plus vite expérimenter le code extrait. Nous avons donc formalisé les premières pages du support de cours de H. Schwichtenberg [73], mais en ne

considérant que les notions nécessaires à la définition de $\sqrt{2}$. Voici par exemple la définition du type des nombres réels :

```
(* First, the Cauchy property. *)

Definition Is_Cauchy (f : nat → Q) (mo : nat → nat) :=
  ∀k m n, mo k ≤ m → mo k ≤ n → let e:=(f m - f n)*2^k in -1≤e≤1.

(* A real is given by a cauchy sequence, a modulus sequence *)
(* and a proof of the Cauchy property of these sequences. *)

Record R : Set := {
  cauchy : nat → Q;
  modulus : nat → nat;
  is_cauchy : Is_Cauchy cauchy modulus }.
```

En fait, même en se limitant ainsi aux notions « utiles », il y en avait pour plus d'une semaine ou deux de travail. Nous avons alors choisi de nous concentrer uniquement sur les portions informatives des termes à définir. Et les parties logiques nécessaires ont alors été posées en axiomes, dans l'attente d'une complétion ultérieure. En réalité, plutôt que de poser de multiples axiomes, ce qui est fastidieux, nous avons en fait « triché », en n'en posant qu'un :

```
Axiom Falsum: False.
Ltac fed_up := elim Falsum.
```

Et l'usage de la tactique `fed_up` permet alors de se débarrasser de toute fin de preuve « agaçante ». Voici par exemple la définition initiale de l'addition de deux nombres réels :

```
Definition Rplus : R → R → R.
 intros x y.
 apply (Build_R (fun n ⇒ cauchy x n + cauchy y n)
                (fun k ⇒ max (modulus x (S k)) (modulus y (S k)))).
 fed_up.
Defined.
```

Dans ce cas, l'usage du `fed_up` permet de se dispenser de la preuve que notre nouvelle suite représentant `x+y` est bien une suite de Cauchy. Et par la suite, ce `fed_up` a bien pu être remplacé par un script d'une trentaine de lignes.

Bien sûr, tant qu'il reste des `fed_up` dans notre développement, on ne pourra pas affirmer avec certitude que la grande fraction ci-dessus est bien une approximation de $\sqrt{2}$ avec la précision mentionnée précédemment. Par contre, l'extraction d'une preuve avec `fed_up` est exactement identique à l'extraction de la même preuve complétée, tant que ces `fed_up` sont utilisés dans des portions de sorte `Prop`. Il convient néanmoins d'être prudent avec cette tactique « magique » :

- S'en servir dans une partie informative revient à placer une exception dans le programme (voir l'extraction de `False_rec` page 79).
- S'en servir pour faire accepter au système une proposition logique erronée peut mener

le programme extrait à un résultat faux, mais aussi à des erreurs d'exécution ou bien à la non-terminaison (voir les exemples du chapitre 2).

6.2.2 Les nombres rationnels

Comme nous connaissions déjà bien les nombres rationnels définis dans FTA, nous avons repris ces rationnels. Mais nous y avons apporté deux améliorations. La première de ces améliorations concerne les preuves impliquant ces rationnels. Lorsque nous avons commencé à remplacer les `fed_up` par de vraies preuves, nous avons en effet constaté que les preuves d'arithmétique rationnelle sont extrêmement pénibles. Le problème provient de la représentation choisie, qui n'est pas canonique : la fraction $\frac{1}{2}$ n'est pas égale à la fraction $\frac{2}{4}$ au sens de l'égalité usuelle de Coq. On doit alors définir une égalité ad hoc sur ce type de données, ce qui prive nous a priori d'un certain nombre d'outils ne fonctionnant qu'avec l'égalité de Coq, comme par exemple des tactiques `rewrite` ou `ring`[8]. Heureusement il existe depuis peu une extension de Coq due à Clément Renard, permettant de travailler plus facilement sur de telles structures dites « sétoïdes » (voir [78]). Nous avons donc équipé notre type `Q` d'une telle structure de sétoïde, ce qui donne accès à des tactiques comme `setoid_rewrite`. Et d'autre part, nous avons contribué à finaliser l'extension de la tactique `ring` vers ces sétoïdes. De la sorte, ces nombres rationnels commencent à être d'un usage raisonnablement pratique, même si un certain nombre d'outils automatiques manquent encore, comme par exemple les tactiques `field` et `fourier`.

La deuxième amélioration concerne plus directement l'extraction. Nous avons en effet remarqué que les opérations sur `Q` ne réduisaient jamais les fractions vers leurs formes canoniques. En conséquence, lors de nos calculs de la constante d'Euler dans FTA, les fractions obtenues grandissent très vite, tout en terminant par exemple par un certain nombre de zéros au numérateur et au dénominateur. Et l'approximation d'ordre 100, qui remplit à l'origine deux pages d'écran, se réduit en fait à une fraction irréductible de quatre lignes seulement. Nous étions au début légèrement réticent à l'idée de simplifier fréquemment les fractions, car cela a également un coût. Mais l'ajout de simplifications dans notre développement expérimental a montré sans ambiguïté possible l'énorme accélération des calculs qui en résulte : on passe en effet de trois décimales accessibles rapidement à plusieurs centaines. Dans le détail, cet ajout de simplification s'est fait via une fonction `Qred : Q→Q`, qui calcule le pgcd du numérateur et du dénominateur, avant de les diviser par ce pgcd. Il ne reste plus alors qu'à prouver que toute fraction retournée par `Qred` est bien équivalente à la fraction d'origine. Nous avons alors inséré un `Qred` dans la boucle principale de calcul de $\sqrt{2}$. Peut-être vaudrait-il mieux en placer à chaque opération élémentaire, ou au contraire moins souvent ? Seuls des tests plus poussés permettront de le dire. En tout cas, il semble évident que les calculs de FTA gagneraient également à intégrer de telles simplifications. Au final, cette bibliothèque améliorée de nombres rationnels a été regroupée dans la nouvelle contribution `Orsay/Qarith`.

[8] Il s'agit d'une tactique automatique pouvant résoudre dans un anneau les égalités impliquant l'associativité, la commutativité et la distributivité de `+` et `*`.

6.2.3 Les suites de Cauchy

Revenons un instant à la définition de `R` donnée précédemment. Cette définition suit la formulation de H. Schwichtenberg [73], qui diffère légèrement de la formulation utilisée par FTA. Cette dernière, plus habituelle, est de la forme $\forall$`k,`$\exists$`N,`$\forall$`n>N,`$\forall$`n>N,|f(n)-f(m)|`$\leq 2^{-k}$, alors que dans [73] la borne `N` est donnée explicitement comme une fonction de `k`, cette dernière fonction étant nommée *module* de la suite de Cauchy. Ces deux formulations sont en fait équivalentes du point de vue constructif, puisque l'on peut retrouver le module `N(k)` à partir de la preuve de $\forall$`k,`$\exists$`N,`... Néanmoins, la formulation plus explicite du module incite à le choisir soigneusement, et nous nous sommes efforcés de le choisir aussi précis que possible. Du coup, lors du calcul d'une approximation de $\sqrt{2}$ par le programme extrait, on obtient directement une majoration de la marge d'erreur du résultat : il suffit de demander `(sqrt2.cauchy (sqrt2.modulus 140))` pour obtenir un résultat proche à 2^{-140} de la limite qu'est $\sqrt{2}$, soit donc 140 chiffres binaires de corrects. En pratique, on obtient même quelques chiffres supplémentaires à cause d'approximations dans le calcul du module de la suite, mais l'ordre de grandeur est le bon. En comparaison, lors des calculs d'approximations de `e` dans FTA, on sent bien que l'approximation de rang `k`, c'est-à-dire le `k`-ième terme de la suite de Cauchy, va nous fournir `n` décimales correctes, mais cette relation entre `k` et `n` n'est pas explicite. Et même si on la rendait explicite, il n'est pas sûr que cette relation serait très précise, vu les choix parfois naïfs des bornes `N` dans FTA. Les estimations d'erreurs des calculs de FTA ont donc été réalisées a posteriori avec Maple.

6.2.4 Les fonctions continues

La façon de définir les fonctions continues diffèrent aussi sensiblement entre [73] et FTA. Dans FTA, une fonction continue est une fonction $\mathbb{R} \to \mathbb{R}$ à laquelle on adjoint certaines propriétés. Vue la définition de $\mathbb{R}$ comme ensemble de suites de Cauchy, une fonction continue dans FTA correspond donc principalement à une fonction de type `(`$\mathbb{N} \to \mathbb{Q}$`)` $\to \mathbb{N} \to \mathbb{Q}$. Ce passage d'une fonction en argument n'est en fait pas souhaitable, car il rend le code extrait plus complexe, plus délicate à analyser et potentiellement moins efficace. L'alternative est alors d'utiliser une famille de fonctions rationnelles qui convergent vers la fonction réelle voulue :

```
Record continuous [i:itvl] : Set := {
   cont_h : Q → nat → Q;
   cont_α : nat → nat;
   cont_w : nat → nat;
   cont_cauchy: ∀a:Q, Is_Cauchy (cont_h a) cont_α;
   cont_unif : ∀a b n k, n≤(cont_α k) → a∈i → b∈i →
       -1 ≤ (a-b)*2^((cont_w k)-1) ≤ 1 →
       -1 ≤ (cont_h a n - cont_h b n)*2^k ≤ 1 }.
```

Dans cette définition, `itvl` est le type des intervalles délimités par deux rationnels. La vraie fonction dans cette définition est `cont_h`, de type $\mathbb{Q} \to \mathbb{N} \to \mathbb{Q}$. Suivent ensuite deux fonctions de modules et leurs propriétés :

- propriété de Cauchy « ponctuelle » quand on fixe le premier argument de `cont_h`.
- uniforme continuité quand on fixe le second argument de `cont_h`.

Ainsi pour la fonction $X \mapsto X^2 - 2$ qui nous intéresse, on prend les valeurs suivantes :

```
Definition sqr2_h := fun (x:Q)(_:nat) ⇒ x*x-2.
Definition sqr2_α := fun _:nat ⇒ 0.
Definition sqr2_w := fun k:nat ⇒ 2+k.
```

6.2.5 Le théorème des valeurs intermédiaires

Enfin, la dernière différence majeure entre ce développement et FTA concerne la méthode utilisée pour trouver la réciproque d'une valeur donnée par une fonction. Il y a en effet deux versions possibles pour le théorème des valeurs intermédiaires (TVI) :

- La première version est le théorème 3.12 de [73]. Il s'agit de la version la plus générale des deux, la seule condition sur la fonction considérée étant d'être continue. Mais la rançon d'une telle généralité est un algorithme coûteux qui divise l'intervalle courant en une multitude de sous-intervalles suffisamment petits, et qui les parcourt avant de s'arrêter à l'un d'entre eux. Accessoirement, il est à noter également que cette version permet de trouver des réciproques approchées aussi finement que l'on veut, mais pas exactes.
- La seconde version est la proposition 3.13 de [73]. Cette version réclame une hypothèse supplémentaire sur notre fonction, qui doit être *localement non-constante* : pour tout sous-intervalle et toute valeur, on peut exhiber un point où l'image de la fonction diffère de cette valeur. Muni de cette information supplémentaire, on peut alors procéder par « trichotomie », variante de la dichotomie adaptée à la logique constructive. Et on obtient ainsi une réciproque exacte.

Compte tenu des avantages de la seconde version en terme d'efficacité et d'exactitude, aussi bien FTA que notre développement l'utilise. Par contre, la distinction se fait au niveau de la preuve de locale non-constance. FTA montre en effet que tout polynôme de degré non-nul est localement non-constant, via une preuve générale basée sur des factorisations complexes des polynômes. Mais pour un cas particulier comme le notre, on peut faire beaucoup plus simple, car la stricte croissance de $X \mapsto X^2 - 2$ sur l'intervalle qui nous intéresse suffit à impliquer que cette fonction est localement non-constante. Et c'est cet algorithme utilisant la stricte monotonie qui produit les bons résultats que nous avons vus.

À l'opposé, L. Cruz-Filipe a établi ultérieurement que le mauvais comportement du calcul de $\sqrt{2}$ par le TVI dans FTA était bien dû à la phase de calcul de locale non-constance sur les polynômes. En utilisant également la stricte monotonie, il a pu obtenir un calcul de $\sqrt{2}$ qui termine. Mais l'efficacité de ce calcul est bien loin de celui de la petite expérimentation présentée ici, ce qui est sans doute dû à des fractions non factorisées et à des calculs sur des termes de preuves faits dans la partie abstraite des réels de FTA.

6.3 Bilan

Arrivé à ce point, il est difficile d'être satisfait par l'état actuel du programme extrait de FTA. Certes, nous avons réussi avec L. Cruz-Filipe à comprendre et optimiser quelques calculs de limites de séries utilisant le formalisme de FTA, mais cette étude a mis en lumière de nombreuses limitations dans le programme d'origine, lesquelles limitations n'ont été supprimées que par une intervention manuelle, parfois très complexe, dans le code d'origine.

Un développement mathématique comme FTA, non pensé à l'origine en terme d'extraction, peut donc occasionner énormément de travail de réécriture avant de pouvoir engendrer un programme raisonnable. L'extraction n'est donc pas un bouton magique créant à coup sûr des programmes intéressants à partir de n'importe quelles preuves. La situation n'est pas si différente des méthodes plus usuelles de développement de logiciel : il ne suffit pas qu'un programme vérifie sa spécification pour qu'il soit un bon programme. Il y a juste ici deux difficultés additionnelles :

- Une analyse a posteriori comme celle qui est en cours pour FTA est très délicate. En particulier l'extraction induit une distance supplémentaire entre le code d'origine à modifier et le programme à analyser, par exemple par profilage.
- Le concept de « bonne preuve » est moins précis que celui de « bon programme ». Du point de vue de l'extraction, une bonne preuve est évidemment une preuve dont l'extraction est efficace. Par contre pour l'utilisateur, une bonne preuve peut être tout simplement une preuve achevée. Et pour le mathématicien, une bonne preuve peut être une preuve élégante ou très abstraite comme celles faites dans `IR`. À l'opposé un développement très spécialisé pour l'extraction, comme celui que nous avons ébauché, sera sans doute qualifié de moins élégant.

L'utilisation du code extrait à partir du théorème principal de FTA est certes pour l'instant un échec. Mais tout d'abord, ce n'est pas un échec définitif, puisque la compréhension de ce code extrait a énormément progressé, et que de nombreuses pistes d'améliorations existent, comme par exemple la simplification des fractions dans FTA. Ensuite, il convient de relativiser cet échec, et en particulier de ne pas en faire l'échec de l'extraction en tant que méthodologie. L'exemple de FTA n'est en effet guère généralisable :

- Il s'agit tout d'abord d'une *première* : aucun développement d'une telle ampleur n'avait encore été extrait. Et lors d'une première, il n'est guère étonnant d'avoir à « essuyer les plâtres ». FTA constitue donc une sorte de mont Everest de l'extraction.
- Il s'agit ensuite d'une *exception* : FTA est le seul développement rencontré par l'extraction ayant une utilisation aussi atypique de la logique de Coq.

Pour finir, il est plutôt rassurant de voir que l'on peut obtenir rapidement des programmes extraits efficaces dans le même domaine que FTA, même si cela suppose de revenir très loin en arrière, et surtout de travailler depuis le début avec l'extraction comme objectif.

Chapitre 7

Une formalisation des ensembles finis

Le développement présenté dans ce chapitre a été réalisé en collaboration avec J.-C. Filliâtre. Les différents fichiers qui le composent, à savoir les sources Coq et les fichiers extraits, sont accessibles sur le site `http://www.lri.fr/~filliatr/fsets`. *Ces fichiers constituent désormais la contribution nommé* `Orsay/FSets`. *Ce développement est également décrit dans l'article conjoint [34], avec un point de vue légèrement différent, moins centré sur l'extraction.*

Pour un programmeur désirant obtenir des programmes certifiés non-triviaux par la méthodologie de l'extraction de programmes, le premier besoin est celui d'une bibliothèque de base certifiée et suffisamment riche pour ne pas devoir réinventer la roue à chaque ligne. Dans cette optique, nous avons pris comme référence la bibliothèque standard accompagnant Ocaml. Est-il possible d'en développer un pendant certifié ?

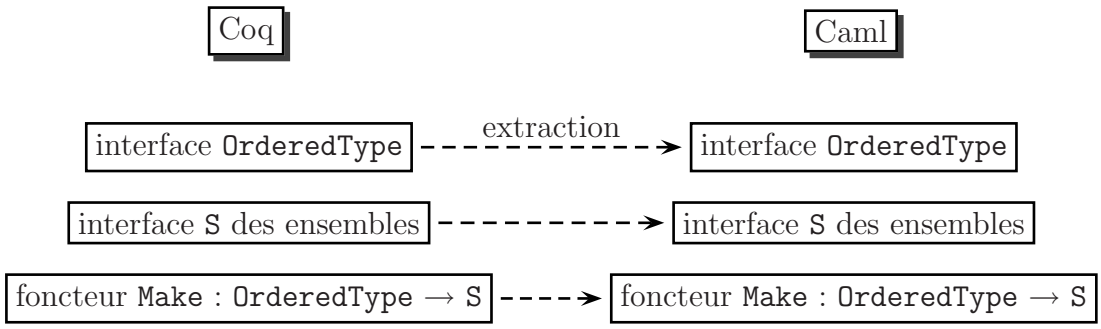
Malheureusement, cette bibliothèque standard d'Ocaml contient peu de structures de données purement fonctionnelles. Or dans notre paradigme d'extraction, l'étude des modules impératifs comme `Array` n'est pas faisable. Pour traiter ces cas impératifs, on se tournera plutôt vers des outils comme Why [33] développé par J.-C. Filliâtre, qui succède à la tactique `Correctness` [32]. D'autre modules, sans être impératifs, sont également de formalisation délicate comme par exemple ceux utilisant les entiers machines.

Il ne reste donc d'intéressant pour notre propos que les modules `List`, `Set`, `Map` et dans une moindre mesure `Sort` et `Stream`. Notre étude s'est finalement limitée au module `Set` des ensembles finis de Ocaml. En même temps, ce choix de `Set` est loin d'être seulement un choix par défaut. Ce module présente en effet les caractéristiques suivantes :

- Le besoin de son utilisation se fait sentir très fréquemment, bien évidemment pour le programmeur, mais également pour le mathématicien. Nous verrons par exemple comment ces ensembles peuvent servir à établir un résultat de la théorie des graphes.
- Son interface, relativement simple, permet de mettre en lumière les nouvelles capacités de Coq en terme d'organisation modulaire. (voir section 4.1).
- La bibliothèque `Set` permet alors de bâtir des modules vérifiant cette interface, grâce à un foncteur `Make` prenant en entrée un module contenant au moins un type et une fonction de comparaison sur ce type. Or il existe de nombreuses manières de coder ce foncteur `Make`, de la plus naïve à la plus raffinée. Par exemple, l'implantation actuellement utilisée par Ocaml est à base d'arbres AVL [2]. Au cours de cette formalisation,

nous avons réalisé une première implantation utilisant des listes triées, avant que J.-C. Filliâtre n'en réalise deux autres plus efficaces à base d'arbres AVL et d'arbres Rouges-Noirs [41].

Le schéma général de notre développement est le suivant :



Une manière simple d'obtenir des interfaces Coq consiste à reprendre les interfaces Ocaml `OrderedType` et `S`, et d'y ajouter uniquement une partie spécification des fonctions. Idéalement, l'extraction de ces interfaces Coq redonnerait alors les interfaces Ocaml originales. Nous avons dans un premier temps suivi cette approche, en dépit d'un certain nombre de difficultés que nous allons détailler dans la section suivante :

- Tout d'abord, des incompatibilités entre les types Ocaml et Coq empêchent l'interface produite par l'extraction d'être exactement égale à l'interface originale de Ocaml.
- Ensuite, la spécification de fonctionnelles d'ordre supérieur comme `fold` s'est révélée particulièrement délicate, et plusieurs versions ont été nécessaires avant d'arriver à un résultat satisfaisant.
- Enfin cette première approche dans laquelle les spécifications sont séparées de la signature des fonctions n'est pas forcément le style le plus naturel en Coq. En effet Coq permet également d'incorporer la spécification dans le type, via l'usage de types dépendants. Nous proposons en fait notre interface Coq sous ces deux versions, ainsi que des passerelles entre ces versions.

De même on peut imaginer importer[1] vers Coq le code actuel du foncteur `Make` de Ocaml, puis dans un deuxième temps prouver que ces fonctions purement calculatoires vérifient bien leurs spécifications séparées. Nous avons suivi cette approche pour la certification d'une version naïve de `Make` à base de listes triées. En revanche J.-C. Filliâtre a préféré suivre l'interface utilisant des types dépendants, et a directement définis les algorithmes et leurs justifications combinées ensemble. La même méthodologie a été utilisé pour obtenir une implantation certifié de `Make` à base d'arbres Rouges-Noirs.

Au final, le programmeur dispose donc de quatre implantations Ocaml compatibles avec l'extraction de notre interface Coq des ensembles :

- l'implantation originale non-certifiée utilisant des AVL, moyennant une légère adaptation manuelle à notre interface ;

[1] Faute de mécanisme plus adapté, cette importation est actuellement à faire manuellement.

- une implantation simple et certifiée mais inefficace à base de listes triées ;
- deux implantation certifiées et efficaces, via des arbres Rouges-Noirs et AVL.

7.1 L'interface Coq

Cette interface correspond au fichier `FSetInterface.v`. Nous allons voir plus en détails les points que Coq permet de traduire directement, et ceux qu'il faut adapter.

7.1.1 Les types ordonnés

Tout d'abord, l'interface Ocaml de `Set` commence par la définition d'une signature représentant un type équipé d'une relation d'ordre, ce type étant destiné à devenir le type support des ensembles.

```
module type OrderedType =
  sig
    type t
      (* The type of the set elements. *)
    val compare : t → t → int
      (* A total ordering function over the set elements.
         This is a two-argument function [f] such that
         [f e1 e2] is zero if the elements [e1] and [e2] are equal,
         [f e1 e2] is strictly negative if [e1] is smaller than [e2], and
         [f e1 e2] is strictly positive if [e1] is greater than [e2].
  end
```

Un premier problème apparaît immédiatement : pour des raisons d'efficacité, la comparaison d'Ocaml retourne un entier machine `int`. Or ces entiers n'existent pas en Coq, et quand bien même, ne seraient pas des plus commodes pour mener des raisonnements logiques. Plus conformément à l'usage en Coq, notre interface est basée sur des relations logiques, c'est à dire des fonctions de type `t→t→Prop`. Ces relations logiques sont l'égalité `eq` et l'ordre strict `lt`. Elles viennent accompagnées de cinq `Axiom` qui assurent que `eq` et `lt` vérifient bien leurs propriétés élémentaires usuelles.

```
Inductive Compare (X : Set) (lt eq : X → X → Prop) (x y : X) : Set :=
  | Lt : lt x y → Compare lt eq x y
  | Eq : eq x y → Compare lt eq x y
  | Gt : lt y x → Compare lt eq x y.

Module Type OrderedType.
  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
```

.../...

```
                                    .../...
  Axiom eq_refl : ∀x, eq x x.
  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬ eq x y.

  Parameter compare : ∀x y, Compare lt eq x y.
End OrderedType.
```

En fait, les versions Ocaml et Coq ne sont pas si différentes, du moins du point de vue informatif, celui de l'extraction. En effet, `eq` et `lt` sont ignorées par l'extraction car placés dans la sorte Prop, celle des propositions logiques. De même, les cinq propriétés de `eq` et `lt` sont également dans `Prop`, donc ignorés. Ne restent donc après extraction que les parties informatives placées dans `Set`, à savoir le type `t`, et la fonction `compare`. Cette dernière fonction, analogue au `compare` d'Ocaml, permet de discriminer selon les positions respectives de deux éléments de type `t`, en retournant un résultat dans le type inductif à trois constructeurs `Compare`. Du point de vue logique, `compare` affirme la décidabilité de `eq` et `lt`. On remarquera que `compare` retourne non seulement l'information de position voulue (via le constructeur utilisé `Lt`, `Eq`, ou `Gt`), mais également la preuve logique qu'on est bien dans cette situation. Cette partie logique est également oubliée par l'extraction, ce qui nous donne pour cette signature `OrderedType` la version extraite suivante :

```
type 'x compare =
  | Lt
  | Eq
  | Gt

module type OrderedType =
 sig
  type t
  val compare : t → t → t compare
 end
```

Il est à noter qu'on peut facilement écrire manuellement des convertisseurs entre ce type `compare` et le type `int` vu comme résultat d'une comparaison à trois valeurs. On peut ainsi adapter le foncteur original `Make` fourni par Ocaml pour qu'il travaille avec notre interface.

7.1.2 La signature des ensembles

Maintenant, en Ocaml, une structure d'ensemble proprement dite est créée via le foncteur suivant :

```
module Make (Ord : OrderedType) : S with type elt = Ord.t
(* Functor building an implementation of the set structure
   given a totally ordered type. *)
```

Et voici maintenant le début de cette signature `S` de la structure d'ensemble :

```
module type S =
  sig
    type elt
    (* The type of the set elements. *)
    type t
    (* The type of sets. *)
    val empty: t
    (* The empty set. *)
    val is_empty: t → bool
    (* Test whether a set is empty or not. *)
    val mem: elt → t → bool
    (* [mem x s] tests whether [x] belongs to the set [s]. *)
    val add: elt → t → t
    (* [add x s] returns a set containing all elements of [s],
       plus [x]. If [x] was already in [s], [s] is returned unchanged. *)
    [...]
  end
```

On y retrouve, outre les types `elt` et `t` et la constante `empty`, 22 fonctions élémentaires sur les ensembles. On notera que chaque fonction est accompagnée d'une spécification informelle en commentaire. C'est en fait à partir de ces spécifications informelles que nous avons bâti notre formalisation. Notre signature Coq commence donc de la même manière que la signature Ocaml, à savoir par la déclaration des types des fonctions ensemblistes :

```
Module Type S.
  Declare Module E : OrderedType.
  Definition elt := E.t.
  Parameter t : Set.
  Parameter empty : t.
  Parameter is_empty : t → bool.
  Parameter mem : elt → t → bool.
  Parameter add : elt → t → t.
  [...]
End S.
```

Quelques signatures de fonctions ne sont pas adaptables aussi simplement :

- `compare : t → t → int`
  On retrouve le même problème que dans `OrderedType` : que faire de `int` ? La réponse est analogue. Tout d'abord la signature `S` contient deux relations `eq` et `lt` sur

les ensembles, de type `t→t→Prop`, sans équivalents en Ocaml, et ignorés par l'extraction. Et à côté de cela, `S` exige la présence d'une fonction informative `compare` de type `∀s∀s',(Compare eq lt s s')`. De la sorte, on a comme en Ocaml la propriété agréable suivante : un module implantant `S` peut également être vu comme un `OrderedType`, ce qui permet la fabrication d'ensembles d'ensembles.

- `iter : (elt → unit) → t → unit`
  Il s'agit de la seule occurrence d'une fonction impérative dans tout le module, intraduisible dans un monde purement fonctionnel, et donc omise.

- `cardinal : t → int`
  Encore une fois, `int` n'est pas utilisable directement en Coq. Nous avons pris ici le parti de déclarer `cardinal : t → nat`, où `nat` est le type Coq des entiers de Peano. Ce choix discutable a le seul intérêt de fournir des principes simples de récurrence sur la taille d'un ensemble. Mais ceci se fait au détriment de l'efficacité. Parmi les autres alternatives, on retrouve les entiers binaires `Z` de Coq, ou encore un type abstrait axiomatisé, que l'on réaliserait à l'extraction dans `int` (voir section 4.4.2). De toute façon, des variantes adaptées de `cardinal` s'écrivent aisément grâce à la fonction générique `fold`.

- `min_elt`, `max_elt` et `choose : t → elt`
  Ces trois fonctions ont en commun de pouvoir lever l'exception `Not_found` si leur argument est vide. La manière naturelle de rendre ce comportement en Coq est l'usage du type `option`.

On remarquera la présence en tête de l'interface Coq d'une déclaration `E : OrderedType`. Cette déclaration, absente en Ocaml, permet en particulier de parler de `E.t`, `E.lt` et `E.eq` dans les spécifications. Le type Coq des modules `Make` sera alors :

```
Module Make (X:OrderedType) : S with Module E := X.
```

Il est possible d'éviter le recours à ce sous-module interne E en le remplaçant par deux `Parameter eq` et `lt` dans `S`, et en fournissant trois « `with Definition ...` » à la place d'un seul « `with Module ...` ».

La partie spécification, deuxième moitié de la signature `S` de Coq, est axée autour d'une relation logique d'appartenance `In : elt→t→Prop`. Cette relation est abstraite : à chaque module implantant nos ensembles finis d'en fournir une. La seule propriété exigée pour ce `In` est la compatibilité vis-à-vis de l'égalité de `E` :

```
Parameter In_1: E.eq x y → In x s → In y s.
```

Cette propriété doit être comprise avec une quantification universelle implicite sur les variables `x`,`y` et `s`, comme l'autorise le mécanisme de `Section` (voir page 30). Il en est de même pour les exemples suivants.

Toutes les spécifications des fonctions ensemblistes sont maintenant exprimées relativement à ce prédicat `In`. Ainsi on écrit par exemple :

```
(** Specification of [mem] *)
Parameter mem_1: In x s → mem x s = true.
Parameter mem_2: mem x s = true → In x s.

(** Specification of [add] *)
Parameter add_1: In x (add x s).
Parameter add_2: In y s → In y (add x s).
Parameter add_3: ¬ E.eq x y → In y (add x s) → In y s.
```

7.1.3 Le cas des fonctions d'ordre supérieur

Parmi ces fonctions ensemblistes, celles qui prennent une fonction en argument nécessitent un peu plus d'attention. Il s'agit de `fold`, `filter`, `for_all`, `exists` et `partition`. Par exemple la spécification informelle du `fold` est :

```
val fold: (elt → 'a → 'a) → t → 'a → 'a
(* [fold f s a] computes [(f xN ... (f x2 (f x1 a))...)], where
   [x1 ... xN] are the elements of [s]. The order in which elements
   of [s] are presented to [f] is unspecified. *)
```

Nos premières tentatives de spécification pour `fold` reposaient sur la formalisation des deux équations suivantes :

```
fold f empty i = i

fold f (add x s) i = f x (fold f s i)
```

Mais cette approche s'est révélée extrêmement dure à finaliser. Deux problèmes se posaient en particulier :

- Pour tenir compte du caractère non-spécifié de l'ordre des calculs, tout en parlant d'un résultat final ayant du sens, il fallait ajouter comme hypothèse sur `f` l'intervertibilité des calculs : `f x (f y a) = f y (f x a)`.
- Que se passe-t-il en outre si `f` retourne deux valeurs distinctes pour deux éléments `x` et `y` par ailleurs égaux modulo `E.eq` ? Dans une version préliminaire, nous ne gérions pas ce cas correctement, ce qui aurait permis de prouver `false = true` à partir d'un hypothétique module ayant pour interface cette version `S`.
- Enfin l'usage de l'égalité usuelle « = » de Coq dans ces équations est trop restrictif pour certains usages. Par exemple si l'on souhaite reconstruire un ensemble via un `fold`, on peut écrire `(fold add s empty)`. Mais l'égalité usuelle ne convient pas à nos ensembles paramétrés par une égalité `E.eq`. On peut alors s'en sortir en utilisant une égalité de plus `eqA` sur le type d'arrivée, mais cela devient vraiment très lourd.

Au final, nous avons opté pour une spécification à la fois plus simple et plus expressive, en s'appuyant sur un type de données déjà connu, à savoir les listes, et plus précisément la fonction `fold_right` définie sur ces listes. Finalement, il s'agit d'une spécification assez proche de la version informelle du commentaire ci-dessus. Au lieu de dire « `x1...xN` sont

les éléments de l'ensemble `s` », on affirme « `l` est une liste sans redondance contenant tous les éléments de `s` et seulement ceux-ci ». Bien sûr, l'appartenance à notre liste et la non-redondance sont à exprimer modulo `E.eq`, ce qui explique la définition de deux prédicats spécialisés `InList` et `Unique` paramétrés par une égalité. Voici donc la spécification définitive de `fold` :

```
Parameter fold_1 : ∀(A : Set)(i : A)(f : elt → A → A), ∃l : list elt,
  Unique E.eq l ∧
  (∀x, In x s ↔ InList E.eq x l) ∧
  fold f s i = fold_right f i l.
```

Cette formulation dispense en particulier de toute pré-condition sur la fonction `f`. Si cette fonction `f` ne vérifie pas l'intervertibilité des calculs, ou bien n'est pas invariante vis-à-vis de `E.eq`, plusieurs listes contenant les éléments de `s` donneront des résultats différents par `fold_right`. Mais l'une de ces listes au moins aboutit au même résultat que le `fold`. En fait, si l'on ajoute ces pré-conditions sur `f`, on peut remplacer le ∃`l` par un ∀`l`.

Pour spécifier les autres fonctions d'ordre supérieur que sont `filter`, `for_all`, `exists` et `partition`, nous aurions pu également établir un parallèle avec les versions de ces fonctions travaillant sur les listes. En fait, ces cas sont sensiblement plus simples que l'exemple de `fold`, puisqu'il n'y a pas de problème d'ordre des calculs ou d'égalité sur le domaine d'arrivée. Nous avons donc utilisé une spécification directe, comme par exemple :

```
Parameter filter_1 : compat E.eq f → In x (filter f s) → In x s.
Parameter filter_2 : compat E.eq f → In x (filter f s) → f x = true.
Parameter filter_3 :
  compat E.eq f → In x s → f x = true → In x (filter f s).
```

La condition `compat` exige alors l'invariance de `f` vis-à-vis de `E.eq`.

7.1.4 Une signature alternative à base de types dépendants

Notre signature est donc divisée en deux, avec d'un côté la fonction purement informative, et de l'autre sa spécification sous forme de lemmes purement logiques. Cette façon de faire n'est pas la seule possible en Coq. Grâce aux types dépendants, on peut en effet tout rassembler en une seule expression, dont le schéma général est alors de la forme, pour une fonction à un argument, `∀x,P(x)→∃y,Q(x,y)` avec `P` et `Q` prédicats logiques exprimant respectivement les pré- et post-conditions.

Nous avons donc écrit une deuxième version de la signature des ensembles, nommée `Sdep`, en utilisant ce style « types dépendants ». En voici un extrait :

```
Module Type Sdep.
  Declare Module E : OrderedType.
  Definition elt := E.t.
                                        .../...
```

.../...

```
  Parameter t : Set.

  Parameter In : elt → t → Prop.
  Definition Empty s := ∀a, ¬ In a s.
  Definition Add (x:elt)(s s':t) := ∀y, In y s' ↔ E.eq y x ∨ In y s.
  [...]

  Parameter empty : {s : t | Empty s}.

  Parameter is_empty : ∀s,{Empty s}+{¬ Empty s}.

  Parameter mem : ∀x s,{In x s}+{¬ In x s}.

  Parameter add : ∀x s,{s' : t | Add x s s'}.
  [...]
End Sdep
```

Le paramètre `In` est maintenant nécessaire dès le début pour exprimer les spécifications. Suivent ensuite un certain nombre de raccourcis tels que de `Empty` et `Add` exprimant des propriétés logiques à base de `In`. La fonction `add` respecte désormais le schéma à base de pré- et post-conditions : la pré-condition est toujours vraie, et la post-condition `(Add x s s')` exprime le fait que le nouvel ensemble `s'` contient les mêmes éléments que l'ancien `s`, plus `x`. Le cas des fonctions `is_empty` et `mem` est un peu différent : au lieu de dire « il existe un booléen tel que ... », nous utilisons directement un type inductif à deux valeurs, sorte de booléen enrichi, permettant d'exprimer ce qui se passe dans les deux cas. Il s'agit du type `sumbool`, présenté p. 29.

Encore une fois, les fonctions les plus difficiles à spécifier sont les fonctions d'ordre supérieur. Voici le `fold`, qui contient la propriété `fold_1` en post-condition :

```
Parameter fold : ∀(A : Set)(f : elt → A → A)(s : t)(i : A),
  {r : A | ∃l : list elt,
    Unique E.eq l ∧
    (∀x, In x s ↔ InList E.eq x l) ∧
    r = fold_right f i l}.
```

7.1.5 Des foncteurs pour choisir son style de signature

Le nouveau système de module de Coq nous permet alors de ne pas avoir à choisir entre les deux signatures possibles. En effet on peut écrire aisément un foncteur qui transforme un module de type `S` en un nouveau module de type `Sdep` et un autre foncteur faisant le travail inverse. Ceci est fait dans le fichier `FSetBridge.v`. De la sorte, toute nouvelle implantation des ensembles n'a besoin que d'être faite en une seule version, peu importe laquelle. Ainsi en fonction du goût du programmeur, les deux implantations à base de listes triées et d'arbres rouges-noirs ont été réalisées l'une conformément à `S`, l'autre conformément à `Sdep`. Et à l'inverse, tout utilisateur a le choix de la version qu'il désire utiliser. Il peut même, et c'est réellement appréciable en pratique, utiliser les deux interfaces simultanément.

7.1.6 Extraction des signatures d'ensembles

Qu'en est-il alors des versions extraites de ces deux interfaces ? Elles sont en fait extrêmement voisines, et peuvent même être rendues égales.

En ce qui concerne l'interface `S`, l'extraction est simple. En effet, les signatures pures ne contiennent aucune partie logique, et il n'y a aucune utilisation de types Coq avancés à base de sortes, de `Cases` ou autres points fixes. L'extraction est alors juste une traduction en Ocaml. Quand aux spécifications, étant complètement logiques, elles sont juste oubliées. On retombe alors sur l'interface Ocaml originale, modulo les adaptions mentionnées précédemment. Voici son début :

```
module type S =
 sig
  module E : OrderedType
  type elt = E.t
  type t
  val empty : t
  val is_empty : t → bool
  val mem : elt → t → bool
  val add : elt → t → t
  [...]
 end
```

Les choses sont plus compliquées dans le cas de l'interface `Sdep`. Tous d'abord, les éventuels arguments logiques correspondant à des pré-conditions sont éliminés. Puis l'inductif correspondant aux post-conditions de la forme `{y: Y| ... }`, à savoir `sig`, est reconnu comme étant un inductif dit « singleton informatif » (voir p. 134), et est donc traduit par l'identité : `type 'a sig0 = 'a`. Le type extrait de `add` en version dépendante est donc `elt → t → t sig0`, qui est donc convertible au `elt → t → t` attendu. Dans l'autre cas de figure, à savoir les fonctions utilisant des `sumbool` comme `is_empty` ou `mem`, l'extraction de `sumbool` consiste à oublier les décorations logiques de cet inductif, ce qui donne :

```
type sumbool =
  | Left
  | Right
```

Ce type `sumbool` extrait est alors isomorphe aux booléens, mais pas égal. Si l'on désire forcer l'égalité, afin d'obtenir réellement la même signature dans les deux cas, il suffit d'utiliser le mécanisme de remplacement des inductifs extraits (voir p. 144) :

```
Extract Inductive sumbool ⇒ bool [true false].
```

Voici le début de l'interface extraite de `Sdep`, sans le remplacement de `sumbool` par `bool` :

```
module type Sdep =
 sig
  module E : OrderedType
  type elt = E.t
  type t
  val empty : t sig0
  val is_empty : t → sumbool
  val mem : elt → t → sumbool
  val add : elt → t → t sig0
  [...]
 end
```

À ce stade, nous avons donc une interface formelle en Coq, ainsi que l'extraction de cette interface en Ocaml. Nous pouvons alors d'ores et déjà fournir une implantation non-formelle de cette interface extraite grâce au module `Set` d'Ocaml, moyennant une encapsulation manuelle de certaines fonctions, notamment les comparaisons.

```
(* c2i : 't compare → int *)
let c2i = function Lt → -1 | Eq → 0 | Gt → 1
(* i2c : int → 't compare *)
let i2c i = if i<0 then Lt else if i=0 then Eq else Gt
(* i2n : int → nat, tail recursive *)
let i2n =
  let rec acc p = function 0 → p | n → acc (S p) (n-1)
  in acc O
module Make(X:OrderedType) : S with module E = X and type elt = X.t =
 struct
  module E = X
  module M = Set.Make(struct
                       type t = X.t
                       let compare x y = c2i (X.compare x y)
                      end)
  include M
  let compare s s' = i2c (compare s s')
  let cardinal s = i2n (cardinal s)
  let max_elt s = try Some (max_elt s) with Not_found → None
  let min_elt s = try Some (min_elt s) with Not_found → None
  let choose s = try Some (choose s) with Not_found → None
 end
```

À noter que ceci ne marche que si on a préalablement remplacé certains types inductifs extraits par leur équivalent prédéfini en Ocaml, à savoir les booléens, les paires et les listes. Pour les booléens, cela peut se faire via un `Extract Inductive`, mais les deux autres cas nécessitent une intervention extérieure, par exemple l'utilisation d'un script d'amélioration syntaxique du code extrait. Ce script, écrit en Camlp4, est disponible à l'adresse suivante : `http://www.lri.fr/~letouzey/download/pp_extract.ml`.

7.2 Une implantation à base de listes triées

Avant d'aller plus loin, on peut s'interroger sur la nécessité de coder en Coq une implantation de nos interfaces d'ensembles. Après tout, en se faisant l'avocat du diable, on peut très bien considérer le module `Set` d'Ocaml comme étant suffisamment éprouvé pour être correct vis-à-vis de sa spécification informelle écrite sous forme de commentaire. Quant à notre foncteur manuel d'encapsulation `Make` du paragraphe précédent, sa petite taille et sa simplicité laissent bien peu de place à des erreurs. Si l'on accepte ces deux points, alors on peut parfaitement réaliser un développement certifié en Coq n'utilisant nos ensembles que via l'interface, et obtenir néanmoins un programme complet grâce au foncteur `Make`.

En fait, nous allons voir plus tard dans la description de notre implantation à base d'arbres AVL que l'implantation d'origine de Ocaml n'était pas si correcte que cela, puisque nous y avons trouvé une erreur. La formalisation de cette implantation n'a donc pas été vaine.

Et du point de vue de l'utilisateur Coq de notre bibliothèque d'ensembles finis, s'arrêter ainsi à l'interface Coq présente deux inconvénients. Tout d'abord une telle vision abstraite exclut tout calcul en Coq. Ainsi on peut bâtir une preuve que `(is_empty empty)` vaut `true`, mais pas calculer/exécuter/simplifier `(is_empty empty)` en `true`. Ceci n'est possible que pour une implantation particulière des ensembles, en attendant peut-être l'introduction de la réécriture dans Coq. Fournir une implantation Coq efficace avec laquelle on peut calculer peut également mener à l'utilisation de ces ensembles par des tactiques basées sur la réflexion (voir par exemple [16]).

L'autre inconvénient est le risque d'incohérence de notre interface, qui n'est en soit qu'une axiomatisation. Ce risque est à prendre au sérieux. Par exemple nous avons déjà mentionné le fait qu'une version initiale de `FSetInterface.v` permettait à partir de certains `OrderedType` de déduire `False`, à cause d'une mauvaise spécification des fonctions d'ordre supérieur comme `fold`. Cette mauvaise surprise ne peut désormais plus se produire, puisqu'au moins une implantation permet, à partir de n'importe quel `OrderedType`, de bâtir un module réalisant `S`, et ce sans recours à aucun axiome.

7.2.1 Description du module `FSetList`

L'objectif de ce module est de fournir le plus rapidement possible une implantation de l'interface Coq des ensembles, principalement afin de vérifier sa cohérence. Il n'y avait donc pas de souci d'efficacité lors de sa création. Du coup, la première idée a été une implantation par des listes quelconques. Mais contrairement aux apparences, les fonctions ensemblistes

sur des listes quelconques ne sont pas si évidentes à écrire. En particulier la fonction `remove` doit parcourir toute la liste pour détecter des doublons éventuels. De même, la fonction `fold` doit gérer correctement les doublons. Une solution est alors de maintenir un invariant de non-redondance. Mais quitte à faire cela, autant prendre directement comme invariant le fait que les listes sont triées : c'est à peine plus compliqué, et beaucoup plus efficace.

Pour cette implantation, nous avons suivi l'interface `S` non-dépendante. Mais il n'est pas simple de travailler avec des listes associées à un invariant. Par exemple, à chaque opération produisant une liste, il faut immédiatement montrer que l'invariant est préservé. Nous avons préféré découper le travail en plusieurs phases.

Un premier foncteur `Raw`, prenant un `OrderedType` en argument, définit des fonctions ensemblistes sur le type de données `t = (list elt)`, tout en faisant comme si ces listes étaient triées. Ainsi l'union correspond à l'algorithme `fusion` classique :

```
Fixpoint union (s : t) : t → t :=
  match s with
    | [] ⇒ fun s' ⇒ s'
    | x :: l ⇒
      (fix union_aux (s' : t) : t :=
         match s' with
           | [] ⇒ s
           | x' :: l' ⇒
             match E.compare x x' with
               | Lt _ ⇒ x :: union l s'
               | Eq _ ⇒ x :: union l l'
               | Gt _ ⇒ x' :: union_aux l'
             end
         end)
  end.
```

Noter également l'astuce usuelle consistant à utiliser un `fix` interne pour permettre l'appel `(union_aux l')`, non structurellement décroissant par rapport au premier argument.

Dans un deuxième temps, le foncteur `Raw` prouve les propriétés attendues par la signature `S`, à ceci près qu'on ajoute en tête des lemmes l'hypothèse que les listes sont initialement triées.

```
Lemma union_1 : ∀(s s' : t)(Hs : Sort s)(Hs' : Sort s')(x : elt),
 In x (union s s') → In x s ∨ In x s'.
```

On prouve également que nos opérations produisent toujours des listes triées si leurs arguments sont triés. Par exemple :

```
Lemma union_sort :
 ∀(s s' : t) (Hs : Sort s) (Hs' : Sort s'),Sort (union s s').
```

Dès lors, nous avons tous les éléments pour définir un deuxième foncteur nommé `Make`, qui cette fois produit réellement un module de signature `S`. Dans ce module, le type de données est maintenant celui des listes bien triées, défini par :

```
Record sorted_list : Set := { this :> Raw.t ; sorted : sort E.lt this }.
Definition t := sorted_list.
```

Le reste du module n'est qu'une longue suite d'encapsulations/décapsulations, qui grâce aux arguments implicites et à la coercion `t :> Raw.t` se font sans problème. Par exemple :

```
Definition union (s s' : t) :=
 Build_sorted_list (Raw.union_sort (sorted s) (sorted s')).
Definition union_1 (s s' : t) := Raw.union_1 (sorted s) (sorted s').
```

7.2.2 Extraction de `FSetList`

L'extraction de tout ceci est des plus simples. Concernant le foncteur `Raw`, les fonctions pures sont extraites en elles-mêmes, et les propriétés sont oubliées. Notre exemple de l'union donne :

```
let rec union s x =
  match s with
    | Nil → x
    | Cons (x0,l) →
        let rec union_aux s' = match s' with
          | Nil → s
          | Cons (x',l') →
              (match E.compare x0 x' with
                 | Lt → Cons (x0,(union l s'))
                 | Eq → Cons (x0,(union l l'))
                 | Gt → Cons (x',(union_aux l')))
        in union_aux x
```

En ce qui concerne le foncteur `Make`, le type `sorted_list` est reconnu comme étant isomorphe à `(list elt)` une fois la partie logique `sorted` disparue (encore un inductif singleton informatif). Tout le reste n'est donc que des définitions d'alias. Ainsi :

```
let this s = s
let union s s' = Raw.union (this s) (this s')
```

7.2.3 La récursivité terminale

Il est à noter que nos fonctions sur les listes ont été écrites dans un style récursif direct, et ne sont donc presque jamais récursives terminales. Ce n'est a priori pas un problème, car ce module n'a pas de prétention d'efficacité. Sauf que ...

Il est en effet apparu lors de la réalisation de l'implantation efficace à base d'arbres Rouges-Noirs, qu'un certain nombre d'opérations, comme par exemple l'union et l'intersection, gagnait à être « sous-traitées » au module `FSetList` :

- Au niveau efficacité, on peut passer d'un arbre Rouge-Noir, qui est en particulier un arbre binaire de recherche, à une liste triée en un simple parcours linéaire. Et la traduction réciproque, bien que beaucoup moins simple, peut également être faite en temps linéaire. Au final, on obtient par exemple une union sur les arbres qui est de complexité au pire la somme des tailles de ses arguments, ce qui est théoriquement optimal, et très loin d'être évident à obtenir par analyse directe des arbres.
- Au niveau de la preuve de correction des fonctions « sous-traitées », il y a un gain patent : il n'y a qu'à prouver une fois pour toutes la correction des deux fonctions de conversion entre les arbres Rouges-Noirs et les listes-triées, et alors la correction des fonctions arboricoles en question s'obtient par translation directe de résultats déjà prouvés sur les listes.

Dès lors, il serait intéressant de réaliser une variante de `FSetList` non consommatrice de pile. Une façon simple d'y parvenir serait sans doute de définir les variantes récursives terminales, et de prouver immédiatement qu'elles retournent le même résultat que les fonctions originales.

En ce qui concerne l'implantation à base d'arbre AVL, nous avons tenu à être le plus fidèle possible au code Ocaml d'origine de la bibliothèque `Set`. Nous n'avons donc pas utilisé cette méthode de mise à plat, de fusion, puis reconstruction, mais la méthode utilisée dans `Set`, plus efficace en pratique.

7.3 Une implantation à base d'arbres Rouges-Noirs

Le fichier correspondant est `FSetRBT.v` Pour mémoire, les arbres Rouges-Noirs (ARN) sont des arbres binaires de recherche dont on limite le déséquilibre maximum en associant deux couleurs aux noeuds, et en contrôlant la disposition de ces couleurs. Plus précisément :

(i) tous les chemins de la racine à une feuille contiennent exactement le même nombre de noeuds noirs.

(ii) un noeud rouge ne peut avoir de fils rouge.

(iii) les feuilles sont considérées noires.

Notre implantation d'ensembles à base d'ARN est donc efficace. Par exemple, la recherche d'un élément a un coût maximal logarithmique. Mais cette efficacité se paie par une complexité des preuves de correction bien plus grande que pour l'implantation précédente. À titre d'exemple, la preuve de correction du `add` fait plus de 350 lignes. Heureusement, seuls `add`, `remove` et `of_list` (la création d'un ARN à partir d'une liste triée) ont présenté de telles difficultés. Les autres fonctions étaient soit plus simples, soit « sous-traitées » au module sur les listes comme on l'a vu au paragraphe précédent.

Notre propos n'est pas ici de décrire les détails de cette implantation, les fonctions principales comme `add` et `remove` ayant d'ailleurs été réalisées par J.-C. Filliâtre. On peut néanmoins noter que l'interface utilisée est `Sdep`, celle à base de types dépendants. Les fonctions sont donc définies par tactiques et non en fournissant directement un terme, et ces tactiques permettent de bâtir à la fois la structure de l'algorithme sous-jacent et de prouver les propriétés intermédiaires nécessaires, les invariants et les post-conditions. Ce module

constitue donc un bon banc d'essai pour l'extraction, qui doit démêler parties informatives et parties logiques. En pratique, les fonctions extraites sont à chaque fois proches de ce qu'on écrirait manuellement, à des détails syntaxiques près comme l'allure des motifs de filtrages. Et comme le montre la section suivante, l'efficacité des fonctions extraites est bien celle attendue.

Prenons comme exemple la fonction `of_list` de construction d'un ARN à partir d'une liste triée. Elle repose sur une fonction `of_list_aux` à trois arguments informatifs :

- la hauteur `k` de l'arbre à fabriquer, initialisée dans `of_list` avec `N_digits`, c'est-à-dire un logarithme en base deux.
- la taille `n` de l'arbre à fabriquer.
- la liste des éléments, qui selon les appels récursifs peut être plus grande que `n`. La fonction `of_list_aux` retourne donc aussi les éléments en surplus.

Voici des versions abrégées de `of_list_aux` et `of_list`. On a supprimé ce qui avait trait aux invariants préservés par `of_list_aux`, pour ne garder que le cheminement des différents cas de figure. Mais même ainsi, ce script Coq reste hautement indigeste, et n'est donné qu'à des fins d'illustration.

```
Definition of_list_aux :
 ∀k : Z, 0 ≤ k →
 ∀n : Z, two_p k ≤ n + 1 ≤ two_p (Zsucc k) →
 ∀l : list elt, sort E.lt l → n ≤ Zlength l →
 {rl' : tree * list elt | ... }.
Proof.
 intros k Hk; pattern k; apply natlike_rec3; try assumption.
 intro n; case (Z_eq_dec 0 n).
 (* k=0 n=0 *)
 intros Hn1 Hn2 l Hl1 Hl2; exists (Leaf,l); [...].
 (* k=0 n>0 (in fact 1) *)
 intros Hn1 Hn2.
 assert (n = 1). [...]
 rewrite H.
 intro l; case l.
  (* l = [],absurd case. *)
  intros Hl1 Hl2; unfold Zlength,Zlt in Hl2; elim Hl2; trivial.
  (* l = x::l' *)
  intros x l' Hl1 Hl2; exists (Node red Leaf x Leaf,l'); [...]
 (* k>0 *)
 clear k Hk; intros k Hk Hrec n Hn l Hl1 Hl2.
 rewrite <- Zsucc_pred in Hrec.
 generalize (power_invariant n k Hk).
 elim (Zeven.Zsplit2 (n - 1)); intros (n1,n2) (A,B) C.
 elim (C Hn); clear C; intros Hn1 Hn2.
                                    .../...
```

```
                                    .../...
 (* 1st recursive call : (of_list_aux (Zpred k) n1 l) gives (lft,l') *)
 elim (Hrec n1 Hn1 l Hl1).
 intro p; case p; clear p; intros lft l'; case l'.
  (* l' = [], absurd case. *)
  intros o; elimtype False.  [...]
  (* l' = x :: l'' *)
  intros x l'' o1.
  (* 2nd rec. call : (of_list_aux (Zpred k) n2 l'') gives (rht,l''') *)
  elim (Hrec n2 Hn2 l''); clear Hrec.
  intro p; case p; clear p; intros rht l''' o2.
  exists (Node black lft x rht, l'''). [...]
Defined.

Definition of_list : ∀l : list elt, sort E.lt l →
 {s : t | ∀x : elt, In x s ↔ InList E.eq x l}.
Proof.
 intros.
 set (n := Zlength l) in *.
 set (k := N_digits n) in *.
 assert (0 ≤ n). [...]
 assert (two_p k ≤ n + 1 ≤ two_p (Zsucc k)). [...]
 elim (of_list_aux k (ZERO_le_N_digits n) n H1 l); auto.
 intros (r, l') o.
 assert (∃n : nat, rbtree n r). [...]
 exists (t_intro r (olai_bst o) H2). [...]
Defined.
```

Et voici maintenant l'extraction de ces deux fonctions :

```
(** val of_list_aux : z → z → elt list → (tree, elt list) prod sig0 **)

let rec of_list_aux x n l =
  match x with
    | ZERO →
        (match z_eq_dec ZERO n with
           | Left → Pair (Leaf, l)
           | Right →
               (match l with
                  | Nil → assert false (* absurd case *)
                  | Cons (x0, l') → Pair ((Node (Coq_red, Leaf, x0,
                      Leaf)), l')))
    | POS p →
        let Pair (n1, n2) = zsplit2 (zminus n (POS XH)) in
        let Pair (lft, l1) = of_list_aux (zpred (POS p)) n1 l in
        (match l1 with
                                    .../...
```

```
                                        .../...
            | Nil → assert false (* absurd case *)
            | Cons (x0,l2) →
                let Pair (rht,l3) = of_list_aux (zpred (POS p)) n2 l2 in
                Pair ((Node (Coq_black,lft,x0,rht)),l3))
    | NEG p → assert false (* absurd case *)

(** val of_list : elt list → t sig0 **)

let of_list l =
  let n = zlength l in
  let Pair (r,l') = of_list_aux (n_digits n) n l in
  r
```

La structure de la preuve de `of_list_aux` se comprend bien mieux en lisant le code extrait que la preuve elle-même, même commentée. Cette lisibilité du code extrait ne va pas de soi, c'est le résultat de nombreuses optimisations (cf. section 4.3). Ici, on reconnaît une récurrence sur l'entier relatif `k≥0`, grâce à un principe de récurrence ad hoc nommé `natlike_rec2`, dont voici le type :

```
Lemma natlike_rec2 :  ∀P : Z → Type,
  P 0 →
  (∀z : Z, 0 ≤ z → P z → P (Zsucc z)) →
  ∀z : Z, 0 ≤ z → P z
```

Dans le cas `k=0`, on a soit `n=0` et alors on fabrique un arbre vide, soit `n=1` et alors on fabrique un noeud terminal rouge. Si maintenant `k>0`, on divise `n-1` en deux, supérieurement et inférieurement (`n1` et `n2`), et on s'appelle récursivement deux fois pour construire les parties gauche et droite de l'arbre.

7.4 Une implantation à base d'arbres AVL

Il s'agit d'une troisième implantation d'un foncteur `Make` prenant un `OrderedType` et retournant un module de signature `Sdep`, après celles utilisant des listes triées et des arbres Rouges-Noirs. Cette implantation a été réalisée par J.-C. Filliâtre en suivant d'aussi près que possible l'implantation d'origine du foncteur `Set.Make` distribué dans la bibliothèque standard d'Ocaml. En fait le code que l'on obtient par extraction de cette implantation Coq est suffisamment proche du code manuel d'origine pour pouvoir raisonnablement affirmer que nous avons formalisé et certifié cette bibliothèque Ocaml.

Hormis les détails d'affichage, la principale différence entre les deux codes concerne l'arithmétique employée. Les arbres AVL sont en effet des arbres dont la différence de profondeur entre les deux sous-arbres ne dépasse pas une certaine quantité Δ fixée[2]. Et Il est alors essentiel de conserver la profondeur de l'arbre dans la structure de données choisie, sous

[2]Dans la littérature, Δ vaut souvent 1, alors que 2 a été choisi par l'implantation Ocaml comme compromis entre les avantages d'avoir des arbres très équilibrés et les coûts liés au rééquilibrage.

peine de recalculer sans cesse cette profondeur. Or une profondeur est un entier machine `int` à l'origine en Ocaml, que nous avons remplacé en Coq par un entier de type `Z`.

Cette différence dans la représentation des entiers a une influence notable sur la vitesse des codes, la version Ocaml manuelle allant environ quatre fois plus vite que la version extraite, comme le montre les chiffres présenté dans [34]. Les entiers Coq encodés via des types inductifs, même sous forme binaire, ne peuvent concurrencer des entiers machines. Il serait donc intéressant de disposer d'un outil permettant de substituer une représentation par une autre lors de l'extraction, généralisant la commande `Extract Inductive`, trop limitée. En théorie, ce remplacement n'est pas sûr, car le type `int` est sensible à des problèmes de dépassement de capacité que ne connaît pas le type `Z` de Coq. Mais ce danger est ici bien hypothétique, puisqu'il parait bien utopique d'espérer manipuler des arbres binaires équilibrés ayant une profondeur supérieure à 2^{30}, sachant qu'ils auraient alors de l'ordre de $2^{2^{30}}$ éléments ! Plus généralement, on peut imaginer remplacer `Z` par `big_int`, qui fournissent une précision arbitraire tout en effectuant un maximum de calcul via les entiers machine.

Au niveau technique, cette implantation est finalement assez similaire de celle sur les arbres Rouges-Noirs, hormis une très grande quantité de raisonnements sur les entiers, et donc une forte utilisation de la tactique automatique `omega`. Nous ne détaillons donc pas, et renvoyons à [34] pour (un peu) plus de détails. La grande surprise a été la découverte d'une erreur dans le code d'origine de Ocaml. Certaines fonctions pouvaient en effet retourner des arbres qui n'était pas correctement équilibrés. Ce problème n'était pas critique dans la mesure où les arbres demeuraient corrects (*i.e.* contenaient toujours les bons éléments), mais par contre l'efficacité pouvait être fortement affectée. Par exemple la complexité logarithmique de la recherche d'un élément, promise dans un commentaire, n'était plus garantie. Ce problème a été signalé à X. Leroy, qui a effectué la correction dans la foulée.

7.5 Un exemple d'utilisation en contexte mathématique

L'exemple qui va suivre est inclus dans notre contribution `Orsay/FSets`, dans le sous-répertoire `PrecedenceGraph`. Ce résultat était à l'origine un exercice posé lors d'un enseignement de théorie des systèmes d'exploitation [9], dont la solution était incorrecte. Ceci nous a amené à rechercher une preuve correcte, et à formaliser cette preuve Coq pour se convaincre une fois pour toute de sa correction. Cette preuve formelle est à l'origine de notre intérêt pour des ensembles finis en Coq. Nous l'avons plus tard modifiée pour qu'elle utilise nos ensembles à la Ocaml. C'est maintenant un bon exemple d'utilisation de ces ensembles dans un cadre mathématique.

Le résultat porte sur les graphes de précédence. Un tel graphe est une représentation sans redondance d'un ordre strict : plus précisément, si $<$ est un ordre strict, le graphe de précédence associé est défini par $a \rightarrow b$ ssi $a < b$ et $\forall c, \neg(a < c < b)$. En pratique, on peut aussi voir un graphe de précédence comme étant un graphe orienté acyclique dont les arcs transitifs ont été supprimés (c'est à dire $a \rightarrow ... \rightarrow b$ implique $\neg(a \rightarrow b)$). Le résultat énonce

alors :

$$E \leq \frac{N^2}{4}$$

avec E étant le nombre d'arêtes d'un graphe de précédence, et N son nombre de sommets.

La preuve se fait par récurrence sur N, en enlevant à chaque fois un sommet bien choisi et les arêtes qui lui sont associées. Cette structure de la preuve nous a poussé à choisir la représentation suivante des graphes[3] :

```
Record Graph : Set := {
  nodes:> t;
  to: nat → nat → bool }.
```

avec `t` étant un ensemble fini d'entiers. De la sorte, lorsqu'on retire un sommet, il n'y a qu'à mettre à jour `nodes`, alors que la fonction de transition `to` peut rester invariante. En effet les seules arêtes qui sont comptées sont celles dont les deux extrémités sont dans `nodes`. La fonction `filter` permet alors de définir les ensembles de successeurs et de prédécesseurs d'un sommet, ce qui en combinaison avec `cardinal` permet de définir le nombre d'arêtes.

La preuve se déroule alors en quatre étapes :

```
Theorem edges_remove : ∀G:Graph,∀n:elt, In n G → to G n n = false →
 nb_edges G = (nb_edges (node_remove G n))+(arity G n).

Theorem get_init : ∀G:AcyclicGraph, 0 < nb_nodes G →
 {p:nat| In p G ∧ nb_pred G p = 0}.

Theorem low_arity: ∀p:nat,∀G:PrecGraph, 0 < nb_nodes G →
 nb_nodes G < 2*(p+1) →
 {k:nat| In k G ∧ nb_linked G k ≤ p}.

Theorem TheBound : ∀G:PrecGraph,
 (nb_edges G)*4 ≤ (nb_nodes G)*(nb_nodes G).
```

Le premier théorème établit que le nombre d'arêtes de $G \setminus \{n\}$ est celui de G moins le nombre de successeurs et de prédécesseurs de n. Ensuite, pour un graphe acyclique, on montre qu'il existe un élément initial. Le troisième théorème est crucial : il affirme que dans tout graphe de précédence de taille strictement inférieure à $2(p+1)$, on peut trouver un sommet avec moins de p voisins. Et c'est ce sommet que nous allons retirer à chaque étape de la récurrence, ce qui nous permet d'établir le théorème final.

7.6 Bilan

Cette étude de cas montre qu'il est parfaitement possible de spécifier et d'implanter des structures de données fonctionnelles et efficaces en Coq, tout en restant en liaison avec une interface et une implantation Ocaml efficace par le biais de l'extraction.

[3]Nous avons choisi d'étiqueter nos sommets avec des entiers. Moyennant un peu plus de travail, tout `OrderedType` pourrait convenir.

Cette démarche peut évidemment être reprise pour d'autres structures, comme celles présentées par C. Okasaki dans [64]. En particulier tout ce travail effectué devrait permettre d'obtenir un module `Map` à faible coût. On peut, en effet, voir une structure de `Map` comme étant un `Set` avec comme `OrderedType` des paires `index * value` dont la comparaison ne porte que sur la première composante. La seule fonction nouvelle à écrire est le `find`, qui n'est de toute façon qu'une variante de `mem`.

Conclusion

Parvenu au bout de ce travail, il est temps de faire le point, en se posant au moins les deux questions suivantes :

- Cette nouvelle extraction est-elle meilleure que l'ancienne ? Nous le pensons.
- Maintenant, notre extraction est-elle encore perfectible ? Nous en sommes certain.

Les acquis

Déjà, les problèmes critiques de correction que nous avions évoqués en introduction ont été réglés. Une preuve syntaxique, permettant de comparer la réduction d'un terme extrait avec celle d'un terme d'origine, assure en effet que des erreurs potentielles d'exécution, du genre de celles dont souffrait l'ancienne extraction, ne peuvent plus se produire à l'heure actuelle, que ce soit avec l'évaluation stricte à la Ocaml ou bien avec l'évaluation paresseuse à la Haskell.

D'autre part nous avons pu finaliser une seconde preuve de correction, inspirée de la réalisabilité, qui garantit que les propriétés sémantiques des termes Coq d'origine sont bien conservées lors de l'extraction. Cette preuve a été faite dans un système aussi proche que possible du Cci actuellement employé dans Coq, ce qui a entraîné un accroissement très conséquent de la complexité de cette preuve sémantique par rapport aux travaux d'origine de C. Paulin. Cette preuve, bien que manuelle et donc en partie insatisfaisante, doit surtout être vue comme une première étape vers une preuve de correction interne, formalisée en Coq, que nous allons évoquer parmi les perspectives.

Une autre réalisation de cette thèse est le règlement du problème du typage des termes extraits. Notre solution consistant à utiliser des fonctions de coercions non-typées est certes ad hoc, mais fonctionne très bien en pratique. Ceci permet dorénavant aux utilisateurs d'explorer toute la gamme des termes Coq sans craindre de ne pas pouvoir extraire du tout ou bien de devoir retoucher à la main leur programme extrait.

Enfin, un effort substantiel a été fait au niveau de l'implantation afin de faire de l'extraction une véritable plate-forme de génération de code certifié, et non plus un outil expérimental. Dans cet ordre d'idée, on notera les gains en lisibilité du code extrait, et surtout les progrès en matière d'intégration de code extrait au sein de développements plus larges, grâce à la génération d'interfaces, et surtout grâce à l'extension de l'extraction au nouveau système de modules de Coq.

Les perspectives

Tout d'abord, il est évident que l'on peut encore progresser dans le domaine de la sécurité de ce mécanisme de génération de code. Bien sûr, comme le montre K. Thompson dans l'article [79], notre programme final ne sera digne de confiance que si chaque maillon de la chaîne le produisant l'est, et en particulier le compilateur Ocaml ou Haskell utilisé, le système d'exploitation, le processeur, etc[4]. En pratique on n'atteint jamais un tel niveau de vérification. Mais l'un des maillons faibles de cette chaîne de sécurité semble bien être aujourd'hui la correction de notre extraction. Il n'est en effet guère satisfaisant que tout ce mécanisme complexe repose uniquement sur une preuve en LaTeX. Il y a alors deux approches possibles pour remédier à cela :

- On peut tout d'abord ajouter au mécanisme d'extraction une extension qui pour chaque terme extrait produit le terme Coq prouvant la correction de ce terme extrait. Dès lors, à chaque extraction, l'acceptation par le vérificateur de types de Coq de ce terme de preuve Coq garantira la correction du terme extrait donné. Cette approche est celle des systèmes comme Minlog ou Isabelle. Et la réalisation d'une telle extension de l'extraction semble à portée de la main, grâce à notre étude de la section 2.4.
- Il existe également une alternative beaucoup plus ambitieuse, mais aussi moins réaliste dans l'immédiat. Il s'agit de la formalisation complète en Coq de notre preuve de correction de l'extraction. Cela demanderait beaucoup plus de travail que l'approche précédente, puisqu'il s'agit d'une formalisation Coq au lieu d'une extension en Ocaml. En même temps on obtient alors une garantie globale, au lieu de construire une preuve particulière à chaque extraction. En ce qui concerne cette éventuelle formalisation complète, on peut imaginer partir de la formalisation de Coq en Coq par B. Barras [8], et réutiliser également l'un des travaux sur la sémantique de ML, comme par exemple [29]. Et pourquoi ne pas envisager alors d'extraire cette formalisation afin de bâtir une extraction faite de code extrait, au moins pour sa partie centrale ? Ceci s'intègrerait naturellement dans le projet encore utopique d'auto-amorçage ou « bootstrap » du noyau de Coq défendu par B. Barras.

Du côté des applications de l'extraction, les travaux sur l'extraction de C-CoRN méritent d'être poursuivis. Bien sûr, de nombreux problèmes persistent encore en matière d'efficacité. Ces problèmes peuvent se situer à plusieurs niveaux. Ainsi les algorithmes employés sont parfois les plus généraux possibles, au détriment de l'efficacité. D'autre part les scripts de preuves sont encore grandement améliorable, pour éviter par exemple tout calcul redondant. Enfin le code généré par l'extraction n'est pas non plus sans défaut. Tout ceci fait qu'il parait difficile d'espérer calculer effectivement des approximations de racines de polynômes via l'extraction de FTA dès les prochains mois. En même temps, si l'on regarde les progrès accomplis, il serait dommage d'en rester là. Rappelons qu'il n'y a pas si longtemps, l'extraction même de FTA semblait irréaliste...

[4]Depuis la rédaction de [79], la situation s'est encore compliquée, puisque la plupart des systèmes modernes utilisent maintenant des bibliothèques dynamiques (`.so` ou `.dll`). Il est donc possible théoriquement de pervertir le comportement d'un programme *après* sa création sans jamais y toucher directement.

Il serait également intéressant de chercher à étendre le domaine d'application de la méthodologie d'extraction. En effet à l'heure actuelle, les programmes adaptés comme candidats à la certification par extraction sont ceux dont seul le résultat de l'exécution importe. Par contre, dans les situations où il est primordial de s'exécuter rapidement ou sans consommer plus d'une certaine quantité de mémoire, alors l'extraction n'est actuellement pas la bonne méthodologie. Ceci exclut immédiatement tout programme embarqué, qu'il soit temps-réel, à mémoire limitée ou les deux. À l'inverse, le domaine d'application actuel contient bien les exemples marquants d'extraction : vérificateurs de tautologies, vérificateurs de typage [8], analyseur de programmes [17]. Peu importe en effet le temps et les ressources utilisés pour vérifier une tautologie ou trouver un contre-exemple, pour typer ou pour analyser un programme, du moment que ce résultat est correct. Actuellement débutent autour de J.-P. Jouannaud des travaux sur le thème de l'évaluation de la complexité en temps des programmes extraits, avec comme modèle certains travaux en Nuprl [11]. Espérons que ces travaux permettront d'étendre le domaine d'application de l'extraction.

Une autre extension possible du domaine d'utilisation de l'extraction consisterait en une internalisation des résultats de programmes extraits en Coq. Après tout, si l'extraction est sûre, pourquoi Coq ne ferait pas confiance à une procédure de décision prouvée puis extraite ? Il y là sans doute matière à interaction avec le compilateur interne de B. Grégoire [40].

Un dernier domaine d'applications pour l'instant hors d'atteinte de l'extraction Coq est l'extraction de programmes à partir de preuves classiques. Il s'agit d'un champ de recherche fort actif, en particulier autour de Minlog. Il faut dire que Coq n'était pas jusqu'à récemment une bonne plate-forme pour de telles études, car l'ajout d'un axiome classique dans `Prop` ne change rien pour l'extraction, alors que l'ajout dans `Set` rend le système incohérent lorsque `Set` est imprédicatif. Or `Set` est maintenant prédicatif par défaut, ce qui rend éventuellement envisageable une extraction classique en Coq.

Enfin, un dernier thème d'amélioration possible est de rendre cette méthodologie plus agréable à utiliser. Un manque actuel est, par exemple, l'impossibilité d'importer en Coq du code ML déjà existant. Bien sûr, on peut toujours traduire à la main, prouver les propriétés que l'on désire, puis extraire, et enfin vérifier que les différence entre le code ML original et celui extrait sont minimes Nous avons procédé de la sorte dans notre certification des ensembles finis d'Ocaml (voir chapitre 7), le code manuel original étant en l'occurrence distribué avec Ocaml. Cette importation de code ML s'apparente aux travaux de C. Parent autour de la tactique `Program` [65], dont l'implantation n'a malheureusement pas pu être adaptée aux versions 7.0 et suivantes de Coq. Il faut bien voir que cette importation est loin d'être évidente. Par exemple l'importation d'une fonction récursive non structurelle va immédiatement nécessiter la preuve justifiant sa bonne fondation. Il serait sans doute intéressant de combiner cette importation avec les travaux d'A. Balaa et d'Y. Bertot facilitant la définition de telles fonctions récursives [6, 5].

On voit donc que les pistes qui s'ouvrent à l'issue de cette thèse sont multiples, même si le chemin parcouru depuis les débuts de l'extraction en Coq il y a quinze ans est déjà considérable. Arrivé à ce point, souhaitons que cette méthodologie de développement de programmes qu'est l'extraction poursuive son essor, en particulier vers le monde industriel. Puisse cette thèse y avoir contribué...

Annexes

A Contributions utilisateurs utilisant l'extraction

- Bordeaux/Additions
- Bordeaux/dictionaries
- Bordeaux/EXCEPTIONS
- Bordeaux/NewSearchTrees
- Bordeaux/SearchTrees
- Dyade/BDDS
- Lannion
- Lyon/CIRCUITS
- Lyon/FIRING-SQUAD
- Marseille/CIRCUITS
- Muenchen/Higman
- Nancy/FOUnify
- Nijmegen/C-CoRN
- Nijmegen/QArith
- Orsay/FSets
- Orsay/QArith
- Rocq/ARITH/Chinese
- Rocq/ARITH/ZChinese
- Rocq/COC
- Rocq/GRAPHS
- Rocq/HIGMAN
- Rocq/MUTUAL-EXCLUSION
- Sophia-Antipolis/Buchberger
- Sophia-Antipolis/Bertrand
- Sophia-Antipolis/Huffman

- Sophia-Antipolis/RecursiveDefinition
- Sophia-Antipolis/Stalmarck
- Suresnes/BDD

Bibliographie

[1] J.-R. Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5) :1259–1263, September 1962.

[3] A. Amerkad, Y. Bertot, L. Rideau, and L. Pottier. Mathematics and proof presentation in pcoq. In *Proceedings of Proof Transformation and Presentation and Proof Complexities (PTP'01)*, 2001. Software available at `http://www-sop.inria.fr/lemme/pcoq`.

[4] D. Aspinall. Proof general : A generic tool for proof development. In *Proceedings of TACAS'2000*, volume 1785. Lecture Notes in Computer Science, 2000. Software available at `http://proofgeneral.inf.ed.ac.uk`.

[5] A. Balaa. *Fonctions récursives générales dans le calcul des constructions*. Thèse d'université, Nice Sophia-Antipolis, November 2002.

[6] A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In Harrison and Aagaard [43], pages 1–16.

[7] H. Barendregt. Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen, 1991. In Handbook of Logic in Computer Science, Vol II.

[8] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

[9] J. Beauquier and B. Bérard. *Systèmes d'exploitation : concepts et algorithmes*. McGraw Hill, 1990.

[10] M. J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1980.

[11] R. Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(1) :3–31, January 2001.

[12] S. Berardi. Pruning simply typed λ-calculi. *Journal of Logic and Computation*, 6(2), 1996.

[13] S. Berghofer. A constructive proof of Higman's lemma in Isabelle. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[14] S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[15] L. Boerio. Extending pruning techniques to polymorphic second order λ-calculus. In *Proceedings ESOP'94*, volume 788. Lecture Notes in Computer Science, 1994.

[16] S. Boutin. Using reflection to build efficient and certified decision procedure s. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281. Lecture Notes in Computer Science, 1997.

[17] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a Data Flow Analyser in Constructive Logic. In D. Schmidt, editor, *European Symposium on Programing, ESOP'2004*, volume 2986 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[18] J. Chrząszcz. Implementing modules in the system Coq. In *16th International Conference on Theorem Proving in Higher Order Logics*, University of Rome III, September 2003.

[19] J. Chrząszcz. *Modules in Type Theory with generative definitions*. PhD thesis, Warsaw University and Université Paris-Sud, 2003.

[20] T. Coquand. An analysis of Girard's paradox. In *Proceedings of the First Symposium on Logic in Computer Science*, Cambridge, MA, June 1986. IEEE Comp. Soc. Press.

[21] T. Coquand and D. Fridlender. A proof of Higman's lemma by structural induction. Technical report, Chalmers University, November 1993. Unpublished draft, available at `ftp://ftp.cs.chalmers.se/pub/users/coquand/open1.ps.Z`.

[22] J. Courant. A Module Calculus for Pure Type Systems. In *Typed Lambda Calculi and Applications 97*, Lecture Notes in Computer Science, pages 112 – 128. Springer-Verlag, 1997.

[23] L. Cruz-Filipe. A constructive formalization of the fundamental theorem of calculus. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, TYPES 2002*, volume 2646 of *LNCS*, pages 108–126. Springer–Verlag, 2003.

[24] L. Cruz-Filipe. *Constructive Real Analysis : a Type-Theoretical Formalization and Applications*. PhD thesis, University of Nijmegen, April 2004.

[25] L. Cruz-Filipe and B. Spitters. Program extraction from large proof developments. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003*, volume 2758 of *LNCS*, pages 205–220. Springer–Verlag, 2003.

[26] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[27] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[28] G. Dowek, G. Huet, and B. Werner. On the Definition of the Eta-long Normal Form in the Type Systems of the Cube. Informal Proceedings of the Workshop "Types", Nijmegen, 1993.

[29] C. Dubois. Typing Soundness of ML within Coq. In Harrison and Aagaard [43], pages 127–144.

[30] H. Benl et al. Proof theory at work : Program development in the Minlog system. In Wolfgang Bibel and Peter H. Schmidt, editors, *Automated Deduction : A Basis for Applications. Volume II, Systems and Implementation Techniques*. Kluwer Academic Publishers, Dordrecht, 1998.

[31] S. Peyton Jones et al. *Haskell 98, A Non-strict, Purely Functional Language*, 1999. Available at `http://haskell.org/`.

[32] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.

[33] J.-C. Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[34] J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In D. Schmidt, editor, *European Symposium on Programing, ESOP'2004*, volume 2986 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[35] H. Geuvers, F. Wiedijk, and J. Zwanenburg. A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals. *LNCS*, 2277 :96–111, 2001.

[36] Herman Geuvers and Milad Niqui. Constructive Reals in Coq : Axioms and Categoricity. *LNCS*, 2277 :79–95, 2001.

[37] E. Giménez. An application of co-Inductive types in Coq : verification of the Alternating Bit Protocol. In *Workshop on Types for Proofs and Programs*, number 1158 in LNCS, pages 135–152. Springer-Verlag, 1995.

[38] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[39] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.

[40] B. Grégoire. *Compilation des termes de preuves : un (nouveau) mariage entre Coq et OCaml*. PhD thesis, Université Paris 7, 2003.

[41] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Ann Arbor, Michigan, 16-18 October 1978. IEEE.

[42] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94 : 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[43] J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics : 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[44] S. Hayashi and H. Nakano. PX, a Computational Logic. Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1987.

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580,583, 1969.

[46] W.A. Howard. The formulae-as-types notion of constructions. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980. Unpublished 1969 Manuscript.

[47] G. Huet, G. Kahn, and Ch. Paulin-Mohring. *The Coq Proof Assistant - A tutorial - Version 8.0*, February 2004. Available at `http://coq.inria.fr/`.

[48] R. Kelsey, W. Clinger, and J. Rees (eds.). *Revised^5 Report on the Algorithmic Language Scheme*, 1998. Available at `http://www.scheme.org/`.

[49] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952.

[50] C. Kreitz. *The Nuprl Proof Development System, Version 5*. Cornell University, Ithaca, NY, 2002. Available at `http://www.nuprl.org`.

[51] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4) :707–723, July 1998.

[52] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3) :269–303, 2000.

[53] X. Leroy, J. Vouillon, D. Doliguez, J. Garrigue, and D. Rémy. *The Objective Caml system – release 3.07*, September 2003. Available at `http://caml.inria.fr/`.

[54] P. Letouzey. Exécution de termes de preuves : une nouvelle méthode d'extraction pour le Calcul des Constructions Inductives. Université Paris VI, 2000. Available at `http://www.lri.fr/~letouzey/download/rapport_dea.ps.gz`.

[55] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[56] P. Letouzey and L. Théry. Formalizing Stålmarck's algorithm in Coq. In Harrison and Aagaard [43], pages 387–404.

[57] Z. Luo. *Computation and Reasoning ; a type theory for Computer Science*, volume 11 of *International Series of Monographs in Computer Science*. Oxford Science Publication, 1994.

[58] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[59] R. Milner. A theory of type polymorphismn programming. *Journal of Computer and System Sciences*, 17, 1978.

[60] J.-F. Monin. Extracting Programs with Exceptions in an Impredicative Type System. In B. Möller, editor, *Mathematics of Program Construction*, volume 947 of *LNCS*. Springer Verlag, 1995.

[61] J.-F. Monin. *Contribution aux méthodes formelles pour le logiciel*. Mémoire d'habilitation à diriger des recherches, Université de Paris Sud, avril 2002.

[62] D. Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998.

[63] C. Murthy and J.R. Russell. A constructive proof of Higman's lemma. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, pages 257–267, 1990.

[64] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

[65] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives*. thèse d'université, École Normale Supérieure de Lyon, January 1995.

[66] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM Press.

[67] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, January 1989.

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

[69] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15 :607–640, 1993.

[70] L. Pottier. Extraction dans le calcul des constructions inductives. In *Journées Francophones des Langages Applicatifs*, 2001.

[71] D. Prawitz. *Natural Deduction, A Proof-Theoretical Study*. Almqvist & Wiksell, 1965.

[72] F. Prost. *Interprétation de l'analyse statique en théorie des types*. PhD thesis, École Normale Supérieure de Lyon, december 1999.

[73] H. Schwichtenberg. Constructive analysis with witnesses. Technical report, Ludwig-Maximilians-Universität, München, 2003. Proceedings of Marktoberdorf '03 Summer School.

[74] M. Seisenberger. *On the Constuctive Content of Proofs*. PhD thesis, Ludwig-Maximilians-Universität München, Fakultät für Mathematik, Informatik und Statistik, 2003.

[75] M. Serrano and P. Weis. Bigloo : a portable and optimizing compiler for strict functional languages. In *2nd Static Analysis Symposium (SAS)*, pages 366–381. Lecture Notes in Computer Science, 1995.

[76] P. Severi and N. Szasz. Studies of a theory of specifications with built-in program extraction. *Journal of Automated Reasoning*, 27(1), 2001.

[77] Julien Signoles. Calcul statique des applications de modules paramétrés. In *Journées Francophones des Langages Applicatifs*, 2003.

[78] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.0*, February 2004. Available at `http://coq.inria.fr/`.

[79] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8) :761–763, Aug 1984.

[80] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics : an Introduction*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1988.

[81] P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language, without even being odd. In *Workshop on Standard ML*, Baltimore, September 1998.

[82] K. Weich. Decision procedures for intuitionistic propositional logic by program extraction. *Lecture Notes in Computer Science*, 1397 :292, 1998. see `http://www.mathematik.uni-muenchen.de/~weich/`.

[83] B. Werner. *Méta-théorie du Calcul des Constructions Inductives*. Thèse d'université, Univ. Paris VII, 1994.

Résumé

Nous nous intéressons ici à la génération de programmes certifiés corrects par construction. Ces programmes sont obtenus en extrayant l'information pertinente de preuves constructives réalisées dans l'assistant de preuves Coq.

Une telle traduction, ou « extraction », des preuves constructives en programmes fonctionnels n'est pas nouvelle, elle correspond à un isomorphisme bien connu sous le nom de Curry-Howard. Et l'assistant Coq comporte depuis longtemps un tel outil d'extraction. Mais l'outil précédent présentait d'importantes limitations. Certaines preuves Coq étaient ainsi hors de son champ d'application, alors que d'autres engendraient des programmes incorrects.

Afin de résoudre ces limitations, nous avons effectué une refonte complète de l'extraction dans Coq, tant du point de vue de la théorie que de l'implantation. Au niveau théorique, cette refonte a entraîné la réalisation de nouvelles preuves de correction de ce mécanisme d'extraction, preuves à la fois complexes et originales. Concernant l'implantation, nous nous sommes efforcés d'engendrer du code extrait efficace et réaliste, pouvant en particulier être intégré dans des développement logiciels de plus grande échelle, par le biais de modules et d'interfaces.

Enfin, nous présentons également plusieurs études de cas illustrant les possibilités de notre nouvelle extraction. Nous décrivons ainsi la certification d'une bibliothèque modulaire d'ensembles finis, et l'obtention de programmes d'arithmétique réelle exacte à partir d'une formalisation d'analyse réelle constructive. Même si des progrès restent encore à obtenir, surtout dans ce dernier cas, ces exemples mettent en évidence le chemin déjà parcouru.

Mots clés. Preuve de programmes. Programmation fonctionnelle. Extraction. Théorie des types. Isomorphisme de Curry-Howard. Calcul des Constructions Inductives. Système Coq.

Abstract

This work concerns the generation of programs which are certified to be correct by construction. These programs are obtained by extracting relevant information from constructive proofs made with the Coq proof assistant.

Such a translation, named "extraction", of constructive proofs into functional programs is not new, and corresponds to an isomorphism known as Curry-Howard's. An extraction tool has been part of Coq assistant for a long time. But this old extraction tool suffered from several limitations : in particular, some Coq proofs were refused by it, whereas some others led to incorrect programs.

In order to overcome these limitations, we built a completely new extraction tool for Coq, including both a new theory and a new implementation. Concerning theory, we developed new correctness proofs for this extraction mechanism. These new proofs are both complex and original. Concerning implementation, we focused on the generation of efficient and realistic code, which can be integrated in large-scale software developments, using modules and interfaces.

Finally, we also present several case studies illustrating the capabilities of our new extraction. For example, we describe the certification of a modular library of finite set structures, and the production of programs about real exact arithmetic, starting from a formalization of constructive real analysis. These examples show the progress already achieved, even if the situation is not perfect yet, in particular in the last study.

Keywords. Proof of programs. Functional programming. Extraction. Type theory. Curry-Howard isomorphism. Calculus of Inductive Constructions. Coq system.