Garantie formelle de correction pour l'extraction Coq

Stéphane Glondu

Stage encadré par Pierre Letouzey
Laboratoire PPS
Université Paris Diderot

Février–Juillet 2007

Le contexte général

Les assistants de preuves permettent de prouver formellement des théorèmes mathématiques « usuels », mais aussi des propriétés sur des programmes et des systèmes informatiques en tous genres. C'est cette étape, et l'étape indissociable de modélisation du comportement du système étudié, que nous appelons *certification*. La certification a occupé beaucoup de chercheurs ces dernières années, et l'utilisation d'assistants de preuve n'est d'ailleurs qu'une méthode parmi d'autres et peut même être combinée avec d'autres. Parmi les assistants les plus connus, on peut citer PVS, Isabelle/HOL, Coq...

Le problème étudié

Nous nous intéressons ici à l'*extraction* de Coq. Les programmes écrits en Coq peuvent contenir beaucoup d'annotations logiques utiles dans la phase de certification, mais totalement inutiles lors de l'exécution du programme. L'extraction consiste en l'effacement de ces annotations, dans le but d'obtenir un programme dans un autre langage (fonctionnel) qui calcule la même chose. Ce dernier point mérite d'être souligné, dans la mesure où la stratégie d'évaluation du langage cible est en général différente de celle de Coq.

L'extraction est actuellement un programme écrit en OCaml, pour lequel aucune preuve formelle n'a été réalisée. Cette extraction est issue des travaux de P. Letouzey, qui a prouvé « sur le papier » des résultats théoriques sur ce processus. L'objectif de ce stage était de voir dans quelle mesure ces résultats peuvent être transposés en Coq.

Cette étape est importante dans le processus de validation de la chaîne de compilation entamé d'une part par B. Barras avec Coq-en-Coq, et d'autre part par Z. Dargaye avec un compilateur de ML certifié.

La contribution proposée

Deux approches ont été abordées.

La première a consisté à écrire une extraction en Coq. B. Barras a, lors de sa thèse, réalisé une implémentation d'un *type-checker* du Calcul des Constructions Inductives — la théorie logique à la base de Coq — en Coq. Ce développement a conduit par extraction à un noyau certifié utilisé par un système de preuves minimal. Comme le souligne B. Barras à la fin de

sa thèse, une formalisation de l'extraction est nécessaire afin de pouvoir réaliser le *bootstrap* d'un système de preuves assimilable à Coq. Nous avons repris les travaux de P. Letouzey dans le formalisme de B. Barras, ajoutant ainsi une extraction au système de preuves certifié déjà réalisé. Cette extraction — ne réalisant pour l'instant aucune optimisation — est pleinement fonctionnelle, et un des principaux théorèmes de P. Letouzey relatif à l'extraction a été prouvé.

L'autre approche vise plus à se greffer sur le système Coq existant, en reprenant l'extraction actuelle, mais en générant en plus du programme extrait une preuve de correction de ce programme. Cela nécessite bien sûr de formaliser le langage cible (ML) en Coq : pour cela, la formalisation de Z. Dargaye a été adoptée et adaptée à nos besoins. Le but est de justifier plus tard toutes les optimisations réalisées lors de l'extraction. L'extraction a donc été réécrite afin de générer des termes plus proches des termes Coq originaux et manipulables en Coq, ceci afin de faciliter la preuve de correction : la correction de fonctions simples sur les entiers issues de cette *extraction interne* peut déjà être établie automatiquement.

Les arguments en faveur de sa validité

Les deux approches, très différentes, ont mené toutes les deux à des résultats encourageants. Dans l'immédiat, l'extraction interne semble plus pragmatique et est réutilisable à peu de frais par les concepteurs d'un compilateur ML certifié, pour obtenir ainsi une compilation certifiée de programmes écrits en Coq. L'approche Coq-en-Coq a permis quant à elle de mettre à l'épreuve une formalisation du système logique de Coq et est une étape importante dans la réalisation d'un noyau totalement certifié.

Le bilan et les perspectives

Nous n'avons traité qu'une extraction très simple, produisant des termes généralement verbeux et ne gérant pas les constantes. Une optimisation des programmes extraits est encore à réaliser avant d'obtenir une extraction comparable à celle présente dans Coq actuellement.

En outre, dans la seconde approche, nous avons restreint notre notion de correction des programmes extraits à la correction de l'évaluation de fonctions totalement appliquées et au type des entiers naturels. Il pourrait être intéressant de généraliser cela et de prouver la correction de l'évaluation de n'importe quelle expression, et de généraliser cette approche à tous les types de données, même ceux contenant des parties fonctionnelles.

La garantie formelle de correction de l'extraction Coq a donc encore du chemin devant elle.

Garantie formelle de correction pour l'extraction Coq

Stéphane Glondu

Stage encadré par Pierre Letouzey
Laboratoire PPS — Université Paris Diderot

Février–Juillet 2007

Résumé

L'assistant de preuves Coq permet la génération de programmes corrects par construction. Cette fonctionnalité, appelée extraction, est de plus en plus exploitée pour produire des bibliothèques de fonctions certifiées, voire des programmes tout entier. Ainsi, un compilateur d'un sous-langage du C a été développé avec Coq, et un compilateur pour langage fonctionnel est en cours.

Coq est maintenant un système très complexe et il a été envisagé d'en réaliser le « bootstrap » : se servir du procédé décrit ci-dessus pour générer par extraction certaines portions cruciales de Coq telles que le type-checker qui lui sert de noyau, ou encore l'extraction elle-même.

Nous étudions ici l'extraction sous deux angles : la génération d'une extraction d'une part, et la certification des programmes extraits d'autre part.

Table des matières

1 Introduction

La certification de programmes

Ces dernières années, le rôle des ordinateurs n'a cessé de croître et maintenant, ces derniers gèrent de nombreux systèmes critiques où la moindre défaillance peut mener au décès de personnes ou à la perte d'importantes sommes d'argent. L'histoire a montré que des erreurs logicielles pouvaient être à l'origine de défaillances graves — l'exemple le plus connu étant probablement l'explosion de la fusée Ariane en 1996.

Depuis la proposition par C. A. R. Hoare en 1969 [7] d'une logique pour prouver des propriétés sur les programmes, beaucoup de chercheurs se sont penchés sur la vérification de programmes. Cette vérification consiste en une première étape de modélisation du comportement souhaité, puis en la preuve que cette modélisation est vérifiée par le programme : c'est la *certification*. Plusieurs types de certifications existent, et plusieurs moyens sont mis en œuvre pour y parvenir.

Actuellement, la programmation impérative est encore généralement préférée dans l'industrie. D'ailleurs, la logique de Hoare traitait essentiellement ce style de programmation. Cependant, les langages fonctionnels sont particulièrement adaptés à la création de programmes certifiés.

Programmation fonctionnelle et λ-calcul

Les langages fonctionnels — tels qu'OCaml, Haskell ou Scheme — sont des langages dérivés du λ-calcul. Or, dès 1958, H. Curry remarque dans [4] une analogie entre le λ-calcul simplement typé et la logique propositionnelle intuitionniste, établissant ainsi un isomorphisme entre types et formules, et entre preuves et programmes. En 1969, W. A. Howard étend cet isomorphisme à la logique du premier ordre en proposant un λ-calcul avec types dépendants [8]. Cet isomorphisme est depuis couramment connu sous le nom d'*isomorphisme de Curry-Howard*.

La formalisation des systèmes logiques a donné naissance à des *assistants de preuves*, logiciels vérifiant la validité de preuves. Avec l'isomorphisme de Curry-Howard en tête, il est tentant de déterminer le contenu calculatoire des preuves, d'*extraire* des programmes à partir des preuves. Les programmes extraits sont alors naturellement exprimés dans des langages fonctionnels. Cette idée a été mise en pratique dans de nombreux systèmes tels que Nuprl [10], Minlog [6] ou Isabelle [2, 3]. Nous nous intéressons ici plus particulièrement à Coq.

Les assistants de preuves au service de la certification

Les assistants de preuves permettent de prouver formellement des théorèmes mathématiques « usuels », mais aussi des propriétés sur des pro-

grammes et des systèmes informatiques en tous genres. Ils sont parfois utilisés pour pour prouver des propriétés sur des programmes impératifs, mais en général seuls certains aspects des langages impératifs sont considérés, et il n'est pas rare de devoir supposer un nombre important d'hypothèses concernant le langage avant de devoir pouvoir prouver des énoncés.

En revanche, les assistants de preuves tels que Coq ou Isabelle permettent de définir des fonctions dans un style fonctionnel et de s'en servir directement dans l'énoncé de théorèmes. Ces fonctions peuvent ensuite se traduire — disons plutôt *s'extraire* — directement vers un langage fonctionnel.

En outre, considérons une preuve (constructive) de l'énoncé suivant :

$$\forall a, b \in \mathbb{N}, \quad b \neq 0 \implies \exists q, r \in \mathbb{N}, \quad a = bq + r \quad \text{et} \quad 0 \leqslant r < b.$$

Derrière cette preuve se cache un programme de division euclidienne, et c'est le rôle de l'extraction de l'exhiber. Le programme (fonctionnel) ainsi obtenu, prenant un a et un b, et retournant un q et un r, vérifiera alors naturellement l'énoncé ci-dessus, pour peu que l'extraction soit correcte.

L'extraction en Coq

Coq est bâti directement sur la correspondance de Curry-Howard : toutes les preuves sont en interne représentées comme des termes du Calcul des Constructions Inductives, un dérivé du λ-calcul. On peut donc dire que toutes les preuves *sont* des programmes. Cependant, dans ces programmes, on peut distinguer des parties purement logiques et des parties vraiment calculatoires, distinctions qui peut déjà être sentie dans certains énoncés : ainsi, dans une proposition existentielle telle que « $\exists x, P(x)$ », on est souvent intéressé par la manière dont le x est construit ; concernant l'énoncé $P(x)$, on a juste besoin de savoir qu'il est vrai, mais la plupart du temps on ne veut pas savoir pourquoi.

L'extraction en Coq a pour rôle de séparer ces différentes composantes, d'effacer les parties logiques afin de ne laisser que les parties calculatoires des preuves. En Coq, cela est réalisé en mettant les propositions purement logiques dans une classe de types particulière. Ainsi, dans l'énoncé « $\exists x, P(x)$ », « $P(x)$ » sera typiquement de type `Prop`, alors que x pourra avoir un vrai type de données. Afin de pouvoir bénéficier de ce qui a été prouvé, il faut quand même garder une relation entre le programme extrait et la proposition de départ. À cette fin, on peut utiliser la réalisabilité, notion introduite par S. C. Kleene en 1945 dans [9].

L'extraction en Coq a commencé avec la thèse de C. Paulin [13, 14, 15], qui a définit une notion de réalisabilité adaptée au Calcul des Constructions, à la base du Coq de l'époque, et une extraction associée. Cette extraction théorique a été prouvée correcte dans [14] et a été implantée dans Coq. Cependant, l'extraction de C. Paulin souffrait de quelques restrictions. Ces restrictions, ainsi que l'évolution de Coq vers le Calcul des Constructions

Inductives, ont mené P. Letouzey à proposer une nouvelle extraction pour Coq dans sa thèse [12]. Cette extraction a été implantée dans Coq, et c'est elle qui est actuellement utilisée.

Une extraction certifiée ?

L'extraction est actuellement un programme écrit en OCaml, pour lequel aucune preuve formelle n'a été réalisée. Cette extraction est issue des travaux de P. Letouzey, qui a prouvé « sur le papier » des résultats théoriques sur ce processus. Cependant, lorsque l'on travaille avec des assistants de preuves, la tentation est grande de prouver de tels résultats en Coq. L'objectif de ce stage était de voir dans quelle mesure ces résultats pouvaient être transposés en Coq.

Cette étape est importante dans la validation de toute une chaîne de compilation entamée d'une part par B. Barras avec le développement en Coq d'un noyau comparable à celui de Coq [1], et d'autre part par Z. Dargaye avec le développement d'un compilateur de ML certifié [5].

Les contributions proposées

Deux approches ont été abordées.

La première a consisté à écrire une extraction en Coq. B. Barras a, dans sa thèse [1], formalisé le Calcul des Constructions Inductives — la théorie logique à la base de Coq — en Coq. Ce développement a conduit par extraction à un noyau certifié utilisé par un système de preuves minimal. Comme le souligne B. Barras à la fin de sa thèse, une formalisation de l'extraction est nécessaire afin de pouvoir réaliser le *bootstrap* d'un système de preuves assimilable à Coq. Nous avons repris les travaux de P. Letouzey dans le formalisme de B. Barras, ajoutant ainsi une extraction au système de preuves certifié déjà réalisé. Un des principaux théorèmes de P. Letouzey relatif à l'extraction a été prouvé. Le développement est disponible sur le web[1].

L'autre approche vise plus à se greffer sur le système Coq existant, en reprenant l'extraction actuelle, mais en générant en plus du programme extrait une preuve de correction de ce programme. Cela nécessite bien sûr de formaliser le langage cible (ML) en Coq : pour cela, une formalisation de Z. Dargaye a été adoptée et adaptée à nos besoins. Le but à terme est de justifier l'extraction, optimisations incluses. Mais pour l'instant, nous avons considéré uniquement une extraction générant des termes le plus proche possible des termes Coq originaux et manipulables en Coq, ceci afin de faciliter la preuve de correction. La correction de fonctions simples sur les entiers issues de cette *extraction interne* peut déjà être établie automatiquement.

[1] `http://www.glondu.net/cgi-bin/darcsweb.cgi?r=stage2007`

L'implémentation est disponible dans la branche `InternalExtraction` (répertoire `contrib/intextr`) du dépôt de sources[2] de Coq.

Bilan

Les deux approches, très différentes, ont mené toutes les deux à des résultats encourageants. Dans l'immédiat, l'extraction interne semble plus pragmatique et est réutilisable à peu de frais par les concepteurs d'un compilateur ML certifié, pour obtenir ainsi une compilation certifiée de programmes écrits en Coq. L'approche Coq-en-Coq a permis quant à elle de mettre à l'épreuve une formalisation du système logique de Coq et est une étape importante dans la réalisation d'un noyau totalement certifié.

Plan

Ce rapport se compose essentiellement de deux parties, qui reflètent les deux approches qui ont été abordées. La section 2 présente la formalisation du CCI en Coq de B. Barras, ainsi que la façon dont nous avons enrichi cette formalisation avec une extraction du CCI écrite en Coq, basée sur la première extraction simplifiée de P. Letouzey. Nous donnons quelques résultats qui ont été prouvés en Coq. La section 3, quant à elle, présente un langage fonctionnel simple (appelé mini-ML) vers lequel la même extraction a été implantée : une commande spéciale, écrite en OCaml, a été ajoutée à Coq. À chaque utilisation, elle engendre un objet Coq représentant le programme mini-ML correspondant à un terme Coq précédemment défini, avec l'objectif de pouvoir générer automatiquement des preuves de correction.

Conventions syntaxiques

Afin de rendre la lecture plus agréable, nous avons choisi d'écrire beaucoup d'énoncés Coq dans un style mathématique. Pour aider le lecteur à mettre en correspondance les énoncés de ce rapport avec les développements Coq, nous donnons les dénominations Coq des définitions et théorèmes en style `machine à écrire`.

2 Coq-en-Coq

Dans sa thèse [1], B. Barras présente une formalisation du Calcul des Constructions Inductives (CCI) — la théorie logique à la base de Coq — en Coq. Son développement aboutit, via l'extraction Coq actuelle, au noyau d'un système de preuves comparable à Coq. Notre objectif était d'ajouter à ce développement une fonction Coq d'extraction pour le CCI, et de prouver quelques propriétés dessus. Dans cette partie, nous présentons le formalisme

[2] `http://gforge.inria.fr/scm/?group_id=269`

de Barras, ainsi que l'extraction qui a été implémentée et les résultats qui ont été prouvés.

2.1 Les Systèmes de Types Purs

Barras conçoit un système de preuves où celles-ci sont représentées par des termes d'un langage typé dérivé du λ-calcul, le *Calcul des Constructions Inductives* (CCI), mettant ainsi en pratique l'isomorphisme de Curry-Howard. C'est, à quelques détails près, le même CCI qui est mis en œuvre dans Coq. Cependant, le CCI et la certification d'un noyau comparable à celui de Coq n'est que l'aboutissement d'un développement commençant avec le λ-calcul et passant par les Systèmes de Types Purs (PTS) avec divers enrichissements. Nous supposerons le lecteur familier avec le λ-calcul typé ainsi qu'avec Coq, et nous présenterons ici rapidement les PTS avec sous-typage et opérateurs (PTSO). Nous invitons le lecteur curieux à consulter les chapitres 5 et 6 de [1] pour plus de détails.

2.1.1 Langage

Les termes sont paramétrés par trois ensembles sur lesquels l'égalité est décidable :

- un ensemble de *sortes* `sort`. Une sorte est une constante particulière, qui est le type d'une certaine classe de types ;
- un ensemble non vide de *noms* `name` (nous noterons `_` un élément particulier de cet ensemble). Ces noms seront notamment utilisés pour le confort de l'utilisateur final (saisie, affichage) pour nommer les variables liées dans un terme, bien que des indices de de Bruijn soient utilisés en interne. Cependant, les noms seront significatifs par la suite lorsque nous introduirons les opérateurs pour exprimer les constantes globales et les inductifs ;
- un ensemble d'*opérateurs* `oper`. Pour éviter d'avoir à trop étendre son langage pour pouvoir tenir compte de toutes les constructions du CCI alors que certaines propriétés ne sont pas spécifiques au CCI, B. Barras utilise cette notion d'opérateur, qui est en quelque sorte une constante équipée d'un schéma de type et éventuellement de règles de réduction.

Définition 2.1 (type `term`)**.** Le type des *termes* est défini par la grammaire suivante :

$$\begin{aligned} T_1, T_2 : \texttt{term} \quad := \quad & s \mid \natural n \mid c \mid c(T_1) \\ & \mid \Pi x : T_1.T_2 \mid \lambda x : T_1.T_2 \\ & \mid T_1 * T_2 \mid (T_1, T_2) \end{aligned}$$

où $s \in \texttt{sort}$, $x \in \texttt{name}$, $c \in \texttt{oper}$, et $n \in \mathbb{N}$.

Remarquons qu'il n'y a pas de construction générique pour l'application, mais qu'il y en a une pour les opérateurs d'arité zéro ou un, les arités supérieures étant simulées à l'aide des paires (T_1, T_2). L'application usuelle sera

ainsi un opérateur prenant en argument un couple (f, x) d'une fonction et d'un argument. Cette présentation permet ainsi de traiter le filtrage d'une façon similaire à l'application. Les sommes $T_1 * T_2$ représentent les types des couples (T_1, T_2), de la même manière que les produits $\Pi x : T_1.T_2$ représentent les types des abstractions $\lambda x : T_1.T_2$. Les constructions $\lambda x : T_1.T_2$ et $\Pi x : T_1.T_2$ sont les seules à lier des variables, et seront utilisées par des opérateurs pour fournir des constructions plus évoluées introduisant des liaisons. Nous rappelons ici que x est purement décoratif, et $\natural n$ représente la variable d'indice de de Bruijn n. De manière générale, nous désignerons par s une sorte, par x un nom et par c un opérateur.

Comme il est de coutume lorsque l'on parle de λ-calcul, on définit l'opération de *substitution* sur les termes :

$$t\,\{\natural 0 \leftarrow u\}$$

désigne le terme t dans lequel toutes les occurrences de $\natural 0$ ont été remplacées par u. Derrière cette description informelle se cache une définition très technique, que nous ne détaillerons pas ici.

Cette notion de substitution va de paire avec la notion d'environnement.

Définition 2.2 (types `env`, `decl`)**.** Un *environnement* est une liste de *déclarations*, et est défini par la grammaire suivante :

$$\Gamma, \Delta : \texttt{env} \quad := \quad [] \mid \Gamma[x : T] \mid \Gamma[x \doteq t : T]$$

Lorsque δ est une déclaration, nous désignerons par x_δ son nom, par T_δ son type et par t_δ son corps (si elle en a un).

Intuitivement, dans un environnement Γ, la variable $\natural 0$ fait référence à la dernière déclaration de Γ.

2.1.2 Signature et typage

Un *jugement de typage* aura la forme $\Gamma \vdash t : T$, signifiant que t a le type T dans l'environnement Γ. t et T sont deux termes du CCI : contrairement au λ-calcul simplement typé, les termes et les types vivent dans le même espace. Dans le même style que les termes, la définition des jugements de typage sera paramétrique, et sera instanciée plus tard au CCI.

Définition 2.3 (type `PTSO_spec`)**.** La spécification d'un PTS avec sous-typage et opérateurs (ou *PTSO*) est une structure regroupant six paramètres :

$$\begin{array}{ll}
\langle\ \texttt{axiom}: & \mathcal{P}(\texttt{sort} \times \texttt{sort}); \\
\quad \texttt{rule}: & \mathcal{P}(\texttt{sort} \times \texttt{sort} \times \texttt{sort}); \\
\quad \texttt{pair}: & \mathcal{P}(\texttt{sort} \times \texttt{sort} \times \texttt{sort}); \\
\quad \leqslant: & \mathcal{P}(\texttt{env} \times \texttt{term} \times \texttt{term}); \\
\quad \Sigma_0: & \mathcal{P}(\texttt{oper} \times \texttt{term}); \\
\quad \Sigma_1: & \mathcal{P}(\texttt{oper} \times \texttt{env} \times \texttt{term} \times \texttt{term} \times \texttt{term}) \\
\rangle &
\end{array}$$

$$\frac{}{[] \vdash}\ \texttt{Wf\_nil} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash T : s}{\Gamma[x : T] \vdash}\ \texttt{Wf\_cons\_var}$$

$$\frac{\Gamma \vdash \quad \Gamma \vdash t : T \quad \Gamma \vdash T : s}{\Gamma[x \doteq t : T] \vdash}\ \texttt{Wf\_cons\_def}$$

TAB. 1 – Règles de typage des contextes – `wf`

Les rôles des différentes composantes devraient devenir clairs avec la prochaine définition. En réalité, la structure `PTSO_spec` de [1] est plus riche et comporte également des preuves de lemmes de compatibilité entre le sous-typage $\leqslant$, la signature Σ_1 et les opérations de substitution et relocation, lemmes que nous ne détaillerons pas ici.

Définition 2.4 (prédicats `wf`, `typ`)**.** Les règles de typage des contextes et des termes des PTSO sont présentées tables 1 et 2.

2.1.3 Sous-typage et réduction

Le sous-typage, utilisé par les règles `Typ_conv` et `Typ_conv_srt`, regroupe dans le cas du CCI les règles de cumulativité et de conversion. C'est donc là qu'interviennent les règles de réduction. En fait, la relation $\leqslant$ donnée par la structure `PTSO_spec` fait appel à des opérations de clôture qui restent génériques. Dans le cadre de l'extraction, on ne s'intéressera guère à l'aspect typage du sous-typage, mais plutôt à l'aspect réduction. Par conséquent, nous ne détaillerons ici que les composantes calculatoires du sous-typage.

Remarque 2.5. Pratiquement toutes les notions présentées dans ce rapport dépendent d'un environnement. Cependant, afin d'alléger les notations, nous omettrons souvent les environnements. Parfois, ces environnements omis seront liés entre eux par des opérations complexes (insertions, relocations, substitutions). En cas de doute, le lecteur pourra se référer au développement Coq.

Définition 2.6 (opération `ctxt`)**.** Soit $\rightarrow$ une règle de réduction. La table 3 défini la *clôture par contexte* $\rightarrow_{\mathcal{X}}$ de $\rightarrow$. Si dans un système, la réduction est close par contexte, elle sera qualifiée de *forte*.

Les `(W)` apparaissant dans la table 3 seront expliqués plus tard, lors de la définition 2.11.

2.2 Le Calcul des Constructions Inductives

Le CCI tel que présenté dans [1] est une instance de PTSO.

$$\frac{\Gamma \vdash \quad (s_1, s_2) \in \texttt{axiom}}{\Gamma \vdash s_1 : s_2}\ \texttt{Typ\_srt} \qquad \frac{\Gamma \vdash \quad \Gamma(n) = \delta}{\Gamma \vdash \natural n :\uparrow^{n+1} T_\delta}\ \texttt{Typ\_rel}$$

$$\frac{\Gamma \vdash \quad [\_ : T] \vdash \quad (c, T) \in \Sigma_0}{\Gamma \vdash c : T}\ \texttt{Typ\_cst0}$$

$$\frac{\Gamma[\_ : T] \vdash \quad \Gamma[\_ : U] \vdash \quad \Gamma \vdash t : T \quad (c, \Gamma, t, T, U) \in \Sigma_1}{\Gamma \vdash c(t) : U}\ \texttt{Typ\_cst1}$$

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma[x : T] \vdash U : s_2 \quad (s_1, s_2, s_3) \in \texttt{rule}}{\Gamma \vdash \Pi x : T.U : s_3}\ \texttt{Typ\_prd}$$

$$\frac{\Gamma[x : T] \vdash M : U \quad \Gamma \vdash \Pi x : T.U : s}{\Gamma \vdash \lambda x : T.M : \Pi x : T.U}\ \texttt{Typ\_lam}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma \vdash B : s_2 \quad (s_1, s_2, s_3) \in \texttt{pair}}{\Gamma \vdash A * B : s_3}\ \texttt{Typ\_sum}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B \quad \Gamma \vdash A * B : s}{\Gamma \vdash (M, N) : A * B}\ \texttt{Typ\_pair}$$

$$\frac{\Gamma \vdash M : U \quad \Gamma \vdash V : s \quad \Gamma \vdash U \leqslant V}{\Gamma \vdash M : V}\ \texttt{Typ\_conv}$$

$$\frac{\Gamma \vdash M : U \quad \Gamma \vdash U \leqslant s}{\Gamma \vdash M : s}\ \texttt{Typ\_conv\_srt}$$

TAB. 2 – Règles de typage des termes – `typ`

$$\frac{t \to t'}{t \to_{\mathcal{X}} t'}\ \texttt{(W)Ctx\_rule} \qquad \frac{t \to t'}{c(t) \to_{\mathcal{X}} c(t')}\ \texttt{(W)Ctx\_cst}$$

$$\frac{T \to T'}{\lambda x : T.M \to_{\mathcal{X}} \lambda x : T'.M}\ \texttt{(W)Ctx\_lam\_l} \qquad \frac{M \to M'}{\lambda x : T.M \to_{\mathcal{X}} \lambda x : T.M'}\ \texttt{Ctx\_lam\_r}$$

$$\frac{T \to T'}{\Pi x : T.U \to_{\mathcal{X}} \Pi x : T'.U}\ \texttt{(W)Ctx\_prd\_l} \qquad \frac{U \to U'}{\Pi x : T.U \to_{\mathcal{X}} \Pi x : T.U'}\ \texttt{Ctx\_prd\_r}$$

$$\frac{A \to A'}{A * B \to_{\mathcal{X}} A' * B}\ \texttt{(W)Ctx\_sum\_l} \qquad \frac{B \to B'}{A * B \to_{\mathcal{X}} A * B'}\ \texttt{(W)Ctx\_sum\_r}$$

$$\frac{M \to M'}{(M, N) \to_{\mathcal{X}} (M', N)}\ \texttt{(W)Ctx\_pair\_l} \qquad \frac{N \to N'}{(M, N) \to_{\mathcal{X}} (M, N')}\ \texttt{(W)Ctx\_pair\_r}$$

TAB. 3 – Règles de passage au contexte – `(w)ctxt`

2.2.1 Sortes

Nous disposons d'une hiérarchie de sortes comprenant deux sortes imprédicatives `Prop` et `Set`[3], et une hiérarchie d'univers prédicatifs $\texttt{Type}_i$ ($i \in \mathbb{N}$). Nous ne détaillerons pas ici les règles de typage des sortes `axiom`, ni celles de formation des produits `rule`.

2.2.2 Opérateurs

C'est là que l'on retrouve toutes les constructions du CCI.

Définition 2.7 (type `cci_op`, instanciation de `oper`)**.** L'ensemble des opérateurs du CCI est le suivant :

$$\begin{array}{rcl} c : \texttt{cci\_op} & := & \pi_1 \mid \pi_2 \mid @ \mid \texttt{Const}\,\{C\} \\ & & \mid \texttt{Ind}\,\{I, n\} \mid \texttt{Constr}\,\{C\} \mid \texttt{Case}\,\{\vec{p}\} \mid \texttt{Fix} \\ & & \mid \square \mid \varepsilon \mid ::\mid \mathcal{M} \mid \mathcal{L} \\ & & \mid \texttt{Record}\,\{\vec{x}\} \mid \texttt{Struct}\,\{\vec{x}\} \mid \texttt{Field}\,\{x\} \end{array}$$

où $n \in \mathbb{N}$, $I, C, x \in \texttt{name}$ et $\vec{p}, \vec{x} \in \texttt{name}^*$.

Nous avons dans l'ordre : les projections, l'application, la constante globale, le type inductif, le constructeur, le filtrage, le point-fixe, cinq opérateurs liés aux *marques*, les enregistrements, l'opérateur correspondant à leur type, et l'accès à un certain champ d'un enregistrement. Les deux dernières lignes ne correspondent pas à des constructions de Coq[4].

Dans la définition précédente, les noms I, C et $\vec{p}$ font référence à un *environnement global* Σ, indexé par des noms. Cet environnement peut contenir des définitions, des axiomes, mais contient aussi la définition des inductifs. En réalité, Barras défini le CCI comme une famille de PTSO (`cci_pts`), indexée par Σ. Dans tous nos développements, nous supposerons Σ fixé, et donc nous travaillerons bien dans un PTSO.

Plutôt que de donner la définition formelle de Σ_0 et de Σ_1 (respectivement `mem_sign0` et `mem_sign` en Coq), nous allons dans la suite de cette section décrire informellement la signification des opérateurs.

(Non-)Curryfication Barras présente son langage d'une façon fortement non curryfiée : là où on s'attendrait à rencontrer plusieurs termes, il n'en met qu'un seul et exploite la construction de paire $(\_, \_)$ des PTSO. Cela permet de simplifier considérablement des énoncés Coq.

[3] Cette hiérarchie de sortes correspond à celle de Coq 6.2. L'imprédicativité de `Set` combinée à d'autres axiomes impliquant une incohérence, elle a été retirée à partir de la version 8.0 : `Set` n'est plus qu'une sorte prédicative au même titre que les $\texttt{Type}_i$.

[4] En Coq, les enregistrements ne sont que du sucre syntaxique pour des inductifs à un constructeur.

Marques Les *marques* — outre leur rôle dans le typage des points fixes que nous ne détaillerons pas — servent canoniquement à marquer une absence ou un délimiteur. $\square$ peut ainsi jouer le même rôle que `()` en OCaml, $\mathcal{M}$ est son type (tout comme `unit`), $\mathcal{L}$ est le type des listes de marques, et ε et :: ses constructeurs. Les quatre premiers de ces cinq opérateurs sont d'arité zéro (ce sont les seuls), le dernier étant unaire.

Notations Écrire les termes formellement en utilisant les opérateurs et les paires peut être très pénible, aussi ne nous hasarderons-nous pas à utiliser systématiquement cette syntaxe[5]. Nous nous permettrons ainsi d'omettre des parenthèses et nous noterons :

- $(t_1, t_2, \ldots, t_n)$, ou $\vec{t}$ (dans un contexte de terme), le terme

$$(t_1, (t_2, \ldots (t_n, \varepsilon) \ldots));$$

- $f\, x$, le terme $@((f, x))$;
- $\texttt{Case}\,\{\vec{p}\}\,(t, P, \vec{f})$, le filtrage de l'objet t en utilisant le prédicat d'élimination P, et les branches sous forme fonctionnelle $\vec{f}$, $\vec{p}$ représentant la liste les noms des constructeurs du type (inductif) de t ;
- $\texttt{Fix}(f : T)$, où f est de la forme

$$\lambda F : T.\lambda p : U.\lambda x : V.M,$$

  la fonction récursive d'argument x de type inductif V (dépendant éventuellement de p de type U), de corps M (les appels récursifs se faisant avec la variable F), le type du point fixe résultant étant T ;
- $\langle \vec{x} \doteq \vec{t} : \vec{T} \rangle$, plus formellement $\texttt{Struct}\,\{\vec{x}\}\,(\vec{T}, \vec{t})$, un enregistrement avec les champs $\vec{x}$ de types $\vec{T}$ et de valeurs $\vec{t}$;
- $\langle \vec{x} : \vec{T} \rangle$, plus formellement $\texttt{Record}\,\{\vec{x}\}\,(\vec{T})$, le type correspondant ;
- $t_{\langle x \rangle}$, plus formellement $\texttt{Field}\,\{x\}\,(t)$, la projection sur x de l'enregistrement t ;

Points fixes Il est à noter que l'argument d'un point fixe ne peut dépendre que d'un seul terme que nous appellerons *paramètre*. L'argument n'est pas non plus obligé de dépendre du paramètre. Cela reviendrait dans Coq à rassembler systématiquement en un seul tous les arguments qui précèdent celui qui décroît structurellement. Le typage des points fixe n'est pas tout à fait le même que dans Coq, mais nous allons ici nous contenter de l'intuition que l'argument du point fixe doit décroître avec chaque appel récursif.

Constantes globales L'opérateur $\texttt{Const}\,\{x\}$ est un opérateur unaire ; la définition de la constante nommée x peut faire référence à un *paramètre* :

[5] D'ailleurs, le système de notations de Coq est beaucoup sollicité dans le développement de B. Barras !

– dans la définition, ce paramètre apparaît sous la forme de $\natural 0$;
– lors de l'utilisation de x, le paramètre correspond à l'argument de $\texttt{Const}\,\{x\}$.

Il ne faut pas confondre $\texttt{Const}\,\{x\}$ avec une fonction, mais plutôt le voir comme une constante paramétrée. L'intuition derrière ce choix est que le paramètre est généralement inférable et — la plupart du temps — ne devrait pas être explicité.

Inductifs L'opérateur $\texttt{Ind}\,\{I, n\}$ est un opérateur unaire, dont l'argument $(p, (a, l))$ contient le *paramètre* (au même sens que Coq) p et l'argument a de l'inductif. n (un entier) et l (une liste de marques) sont utilisés pour le typage des points fixes. La définition de l'inductif[6] est identifiée par I dans l'environnement global Σ. C'est dans Σ qu'est stockée la liste des constructeurs de I, ainsi que leurs types.

Constructeurs L'opérateur $\texttt{Constr}\,\{C\}$ est un opérateur unaire dont l'argument (p, a) contient le paramètre p de l'inductif associé et l'argument a du constructeur.

Remarque 2.8 (Distinction `Set`/`Prop`)**.** La sorte `Prop` est utilisée pour typer les propositions logiques, et la sorte `Set`, pour typer les types de données d'objets calculatoires. Intuitivement, l'utilité des propositions est de savoir qu'elles sont prouvées ; un objet calculatoire ne doit pas dépendre de la façon dont une proposition a été prouvée. Ce fait est d'ailleurs exploité dans l'extraction : toutes les preuves de propositions logiques seront effacées. Cela impose certaines contraintes sur les filtrages, que nous ne détaillerons pas ici.

2.2.3 Réduction

Pour avoir une instanciation complète de PTSO, nous devons aussi donner la règle de sous-typage $\leqslant$. Cette règle en soit joue un rôle mineur dans l'extraction, donc nous ne la présenterons pas ici. Cependant, elle contient la relation de réduction $\rightarrow_{\mathcal{C}}$ qui nous servira à établir la correction de l'extraction.

Définition 2.9 (prédicats `redn_term`, $\rightarrow_{\mathcal{C}}$)**.** La règle `redn_term` est l'union des règles détaillées table 4. $\rightarrow_{\mathcal{C}}$ est la clôture par contexte de `redn_term`.

Petit point sur les notations « $\Gamma \vdash t \rightarrow_{\mathcal{C}} u$ » désigne ce que l'on note en Coq « `ctxt redn_term` $\Gamma\ t\ u$ ». Il s'agit de *la* règle correspondant à une étape de réduction du CCI.

[6] Les inductifs mutuels et les points fixes mutuels sont traités dans le développement Coq, mais nous écarterons silencieusement ces possibilités ici.

$$\frac{\Gamma(n) = [x \doteq t : T]}{\Gamma \vdash \natural n \to_\delta \uparrow^{n+1} t}\ \texttt{delta} \qquad \frac{\Sigma(x) = [x \doteq t : T]}{\texttt{Const}\,\{x\}\,(M) \to_\Delta t\,\{\natural 0 \leftarrow M\}}\ \texttt{delta\_glob}$$

$$\frac{}{(M,N) \to_{\pi_1} M}\ \texttt{proj1} \qquad \frac{}{(M,N) \to_{\pi_2} N}\ \texttt{proj2}$$

$$\frac{y = x_i}{\langle \vec{x} \doteq \vec{t} : \vec{T} \rangle_{\langle y \rangle} \to_{\pi_r} t_i}\ \texttt{projr}$$

$$\frac{}{(\lambda x : T.M)N \to_\beta M\,\{\natural 0 \leftarrow N\}}\ \texttt{beta}$$

$$\frac{t = \texttt{Constr}\,\{C\}\,(M)}{\texttt{Fix}(f : T)\ p\ t \to_{\iota_f} f\ (\texttt{Fix}(f : T))\ p\ t}\ \texttt{iota\_fix}$$

$$\frac{t = \texttt{Constr}\,\{C\}\,(M) \qquad p_i = C}{\texttt{Case}\,\{\vec{p}\}\,(t, P, \vec{f}) \to_{\iota_c} f_i\,M}\ \texttt{iota\_case}$$

TAB. 4 – Réduction dans CCI – `redn_term`

2.3 Extraction vers CCI$_\Box$

L'extraction consiste à transformer un terme CCI en un terme ML (par exemple) en effaçant toutes les annotations de type, ainsi que les preuves de propositions logiques. Cette opération ne devrait pas changer la sémantique du terme s'il calcule effectivement quelque chose.

On pourrait être tenté de définir directement une fonction d'extraction d'un terme CCI vers un terme ML. Cependant, comme l'explique Letouzey dans [12], une telle fonction ne présente pas de bonnes propriétés et cette approche n'est pas suffisamment générale si l'on veut prouver la correspondance entre les réductions du terme original et celles du terme extrait. Par conséquent, nous utiliserons plutôt une relation pour modéliser une extraction « partielle ».

Définition 2.10 (prédicat `Pe`)**.** La relation d'extraction $\Gamma \vdash t \to_\mathcal{E} t'$ est définie par les règles de la table 5. On dira alors que t s'*extrait* (partiellement) vers t' dans le contexte Γ.

Dans la table 5, on peut distinguer trois types de règles :
- `Pe_Prop` élimine les parties logiques ;
- les autres règles produisant $\Box$ éliminent les types ;
- les règles restantes ne font que propager l'extraction aux sous-termes, en prenant soin d'effacer les types.

La relation $\to_\mathcal{E}$ n'est pas fonctionnelle : un chevauchement est fort possible entre les règles.

Comme mentionné à la remarque 2.5, nous omettrons souvent le « $\Gamma \vdash$ ». Notons que le terme t' reste syntaxiquement un terme de CCI, mais appar-

$$\frac{\Gamma \vdash t : T : \texttt{Prop}}{\Gamma \vdash t \to_{\mathcal{E}} \square}\ \texttt{Pe\_Prop} \qquad \frac{t \in \texttt{sort} \cup \texttt{oper}}{t \to_{\mathcal{E}} \square}\ \texttt{Pe\_\{Srt,Cst0\}}$$

$$\frac{}{t_1 * t_2 \to_{\mathcal{E}} \square}\ \texttt{Pe\_Pair} \qquad \frac{}{\Pi x : T.t \to_{\mathcal{E}} \square}\ \texttt{Pe\_Prd} \qquad \frac{}{\natural n \to_{\mathcal{E}} \natural n}\ \texttt{Pe\_Rel}$$

$$\frac{c \in \{\texttt{Ind}\,\{I, n\}, \texttt{Record}\,\{\vec{x}\}\}}{c(t) \to_{\mathcal{E}} \square}\ \texttt{Pe\_\{MutInd,Record\}}$$

$$\frac{t \to_{\mathcal{E}} t' \quad i \in \{1, 2\}}{\pi_i(t) \to_{\mathcal{E}} \pi_i(t')}\ \texttt{Pe\_Proj}i$$

$$\frac{t_1 \to_{\mathcal{E}} t'_1 \qquad t_2 \to_{\mathcal{E}} t'_2}{t_1\, t_2 \to_{\mathcal{E}} t'_1\, t'_2}\ \texttt{Pe\_App} \qquad \frac{t_1 \to_{\mathcal{E}} t'_1 \qquad t_2 \to_{\mathcal{E}} t'_2}{(t_1, t_2) \to_{\mathcal{E}} (t'_1, t'_2)}\ \texttt{Pe\_Pair}$$

$$\frac{t \to_{\mathcal{E}} t'}{\lambda x : T.t \to_{\mathcal{E}} \lambda x : \square.t'}\ \texttt{Pe\_Lam} \qquad \frac{f \to_{\mathcal{E}} f'}{\texttt{Fix}(f : T) \to_{\mathcal{E}} \texttt{Fix}(f' : \square)}\ \texttt{Pe\_Fix}$$

$$\frac{t \to_{\mathcal{E}} t' \qquad f \to_{\mathcal{E}} f'}{\texttt{Case}\,\{\vec{p}\}\,(t, P, f) \to_{\mathcal{E}} \texttt{Case}\,\{\vec{p}\}\,(t', \square, f')}\ \texttt{Pe\_Case}$$

$$\frac{t \to_{\mathcal{E}} t' \qquad c \in \{\texttt{Const}\,\{x\}, \texttt{Constr}\,\{C\}\}}{c(t) \to_{\mathcal{E}} c(t')}\ \texttt{Pe\_\{Const,Constr\}}$$

$$\frac{t \to_{\mathcal{E}} t'}{\langle \vec{x} : T := t \rangle \to_{\mathcal{E}} \langle \vec{x} : \square := t' \rangle}\ \texttt{Pe\_Struct} \qquad \frac{t \to_{\mathcal{E}} t'}{t_{\langle x \rangle} \to_{\mathcal{E}} t'_{\langle x \rangle}}\ \texttt{Pe\_Projr}$$

TAB. 5 – Prédicat d'extraction – Pe

$$\frac{}{\texttt{Fix}(f : T)\ p\ \square \to_{\square_f} f\ (\texttt{Fix}(f : T))\ p\ \square}\ \texttt{dummy\_fix}$$

$$\frac{}{\texttt{Case}\ \{\vec{p}\}\ (\square, P, \vec{f}) \to_{\square_c} f_1\ \square}\ \texttt{dummy\_case}$$

TAB. 6 – Règles de réduction des $\square$

tient à un sous-langage de CCI, que nous noterons $\text{CCI}_\square$. C'est ce qui jouera le rôle de notre ML. Les $\square$ introduits représentent les parties éliminées par l'extraction. Il est important de remarquer que CCI et $\text{CCI}_\square$ ne partagent que de la syntaxe : en général, t' n'est pas un terme correctement typé du CCI. De plus, nous allons munir $\text{CCI}_\square$ de règles de réductions différentes de celles du CCI, afin de mieux correspondre à ML.

2.4 Réduction dans $\text{CCI}_\square$

Nous décrivons ici les réductions du $\text{CCI}_\square$. Ces réductions correspondent à un langage de programmation fonctionnelle tel que Caml. Contrairement au CCI, les sous-termes d'une abstraction ne sont pas réduits en $\text{CCI}_\square$, ce qui motive la définition suivante :

Définition 2.11 (opération `wctxt`)**.** Soit $\to$ une règle de réduction. La *clôture par contexte faible* $\to_{\mathcal{W}}$ est définie comme dans la table 3, en enlevant les règles `Ctx_lam_r` et `Ctx_prd_r`.

De plus, si on regarde la table 4, la règle `iota_case` ne peut plus convenir lorsque l'objet filtré est logique (*i.e.* de sorte `Prop`) : en effet, l'objet filtré peut devenir $\square$ après extraction (et le deviendra toujours si l'extraction est poussée jusqu'au bout). Cependant, pour qu'un filtrage sur un inductif logique intervienne dans un « véritable » calcul, il faut qu'il ait exactement un constructeur dont l'argument est purement logique, en vertu de la remarque 2.8. Un problème similaire se présente avec les points fixes dont l'argument de décroissance est logique (`iota_fix`). C'est pourquoi nous introduisons table 6 les deux règles de réduction `dummy_fix` et `dummy_case`.

Définition 2.12 (prédicats `dummy_redn_term` et $\to_\square$)**.** L'union des règles détaillées tables 4 et 6 forme la règle `dummy_redn_term`. $\to_\square$ est la clôture par contexte faible de `dummy_redn_term`.

Petit point sur les notations « $\Gamma \vdash t \to_\square u$ » désigne ce que l'on note en Coq « `wctxt dummy_redn_term` $\Gamma\ \ t\ \ u$ ». Il s'agit de *la* règle correspondant à une étape de réduction du $\text{CCI}_\square$.

2.5 Propriétés de $\rightarrow_{\mathcal{E}}$

Cette section est une application assez fidèle de l'étude syntaxique menée par P. Letouzey dans la section 2.3 de [12] au CCI de B. Barras.

Nous allons énoncer des propriétés de $\rightarrow_{\mathcal{E}}$ qui ont été prouvées en Coq lors de ce stage. Les présentations des lemmes ci-dessous feront appel à des notions et notations qui ne seront pas définies formellement dans ce rapport, mais juste expliquées informellement. Bien sûr, ces notions ont été formellement définies en Coq.

Lemme 2.13 (`Pe_weak`)**.** *La règle suivante est admissible :*

$$\frac{\Gamma\Delta \vdash t \rightarrow_{\mathcal{E}} t' \qquad \Gamma\delta \vdash}{\Gamma\delta\Delta' \vdash \uparrow^1_{|\Delta|} t \rightarrow_{\mathcal{E}} \uparrow^1_{|\Delta|} t'}$$

Cette règle dit que l'extraction est invariante par ajout d'une déclaration δ dans un environnement $\Gamma\Delta$. $\uparrow^n_k t$ est une notation pour l'opération — bien connue lorsque l'on travaille avec des indices de de Bruijn — de *relocation*, où toutes les variables libres sous k lieurs de t sont incrémentées de n. Δ' représente l'environnement Δ ayant subit les opérations idoines de relocation. On remarquera la ressemblance entre ce lemme et le lemme d'*affaiblissement* que l'on rencontre en typage[7]. Le script de preuve en Coq est d'ailleurs quasiment identique.

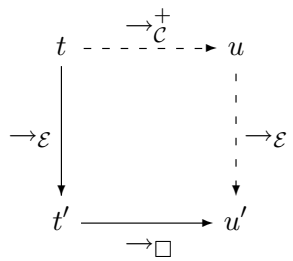
Le lemme suivant est aussi similaire à un lemme de typage, le lemme de *substitution*[8] :

Lemme 2.14 (`Pe_sub`)**.** *La règle suivante est admissible :*

$$\frac{\Gamma[x:U] \vdash t : T \qquad \Delta \vdash u : U \qquad t \rightarrow_{\mathcal{E}} t' \qquad u \rightarrow_{\mathcal{E}} u'}{t\left\{\natural 0 \leftarrow u\right\} \rightarrow_{\mathcal{E}} t'\left\{\natural 0 \leftarrow u'\right\}}$$

Le théorème suivant est un des principaux résultats de [12] — il s'agit du théorème 2. Il exprime que l'on peut simuler au niveau CCI toute réduction d'un terme extrait.

Théorème 2.15 (`Pe_wctxt_correct`)**.** *Soient t un terme CCI clos bien typé et t', u' deux termes de $CCI_{\square}$ tels que $t \rightarrow_{\mathcal{E}} t'$ et $t' \rightarrow_{\square} u'$. Il existe alors un terme CCI u tel que $u \rightarrow_{\mathcal{E}} u'$ et $t \rightarrow^+_{\mathcal{C}} u$.*



[7] lemme 5.25 `typ_weak` de [1], page 140
[8] lemme 5.26 `typ_sub` de [1], page 140

En Coq, cela donne :

```
Theorem Pe_wctxt_correct : forall e t t' u' T,
  closed 0 t ->
  gclosed t ->
  sg -- e |- t : T ->
  Pe e t t' ->
  wctxt dummy_redn_term e t' u' ->
  exists u, R_t (wctxt redn_term) e t u /\ Pe e u u'.
```

La preuve suit le même principe que dans [12] : on procède par cas selon la réduction employée entre t' et u'. Pour chaque redex dans t', on trouve un redex correspondant dans t. La preuve en Coq est longue — un peu plus de 350 lignes — et pénible par moments, notamment à cause de la représentation des termes avec opérateurs qui laisse la possibilité de nombreux termes absurdes (tels que l'opérateur @ appliqué à autre chose qu'un couple). Pour éliminer ces termes mal formés, nous avons utilisé le typage, alors que l'on pourrait s'attendre à n'utiliser le typage que pour traiter la règle `Pe_Prop`. Il est important de noter que les termes extraits ne sont pas typés : le fait que les termes extraits sont issus de termes bien typés est alors crucial pour justifier leur « bonne forme ».

Dans l'hypothèse où le CCI normalise fortement, on prouve facilement la normalisation forte des termes extraits.

Corollaire 2.16 (`Pe_redn_sn`)**.** *Soit t un terme clos bien typé dans CCI et t' tel que $t \to_{\mathcal{E}} t'$. Alors toute suite de dérivations $\to_{\square}$ partant de t' est finie.*

Finalement, nous avons montré qu'un calcul sur un terme extrait termine toujours sur une réponse reliée au terme CCI correspondant. Par contre, nous n'avons aucune garantie que le calcul ne bloque pas trop tôt, sur un terme non totalement réduit vers une valeur.

2.6 Une extraction extraite

Nous avons ajouté au système de preuves de Barras une commande d'extraction en implémentant une fonction `extract_term` vérifiant pour tout terme t clos bien typé la propriété :

$$t \to_{\mathcal{E}} \texttt{extract\_term}(t)$$

Cette fonction est essentiellement une déterminisation de la relation $\to_{\mathcal{E}}$ privilégiant les règles d'élagage, afin d'obtenir l'extraction la plus précise possible.

À titre d'illustration, voici une session de l'interpréteur mettant en œuvre l'extraction :

```
Bcoq < Inductive nat: Set := O: nat | S: (!nat _ ~)->nat.
nat defini(s) inductivement.

Bcoq < Definition plus: nat -> nat -> nat :=
  (Fix {X, F | (nat-X)->nat->nat :> nat->nat->nat :=
       [n:nat+X]<[a:Lmark][_:(!nat ~ a)]nat->nat>Cases n of
          | O => [_:Lmark][m:nat]m
          | S => [pn:(nat-X)*Lmark][m:nat](S (F^0 ~ pn^0 m))
          end}^0 ~).
plus defini.

Bcoq < Print plus.
plus:
(P1 Fix((Lmark->nat->nat->nat)*Lmark,[X:Mark]
  [F:(Lmark->Ind{nat;0}(~,(~,::(X,~)))->nat->nat)*Lmark]
  ([_:Lmark][n:Ind{nat;1}(~,(~,::(X,~)))]Case{O|S}(n,([a:Lmark]
    ([_0:Ind{nat;0}(~,(a,~))]nat->nat,([_0:Lmark][m:nat]m,
      ([pn:Ind{nat;0}(~,(~,::(X,~)))*Lmark][m:nat]
        Cstr{S}(~,(P1 F ~ P1 pn m,~)),~)))),~)) ~)

Bcoq < Extraction plus.
Extraction de plus:
(P1 Fix(#,[X:#]
  [F:#]
  ([_:#][n:#]Case{O|S}(n,(#,
    ([_0:#][m:#]m,
      ([pn:#][m:#]
        Cstr{S}(#,(P1 F # P1 pn m,#)),#)))),#)) #)
```

L'invite de l'interpréteur est `Bcoq <`, et chaque commande entrée par l'utilisateur se termine par un point. Dans cette session, le type des entiers est défini, puis l'addition sur les entiers. La syntaxe est encore lourde, et nous avons rajouté manuellement des sauts de lignes et l'indentation sur les sorties de `Print plus` et de `Extraction plus` afin de les faire correspondre.

3 Extraction interne

Nous décrivons dans cette partie le deuxième grand axe qui a été exploré lors du stage. Cette partie — beaucoup plus expérimentale — a été abordée surtout en fin de stage. Aussi adopterons-nous une syntaxe beaucoup plus proche de Coq.

3.1 L'extraction dans Coq

Actuellement, l'extraction de Coq (implémentée en OCaml) transforme un terme Coq en un programme (sous forme de chaîne de caractères) OCaml, Haskell ou Scheme. Cependant, aucune garantie formelle n'est donnée quant à la correction des programmes extraits, et donc un bug dans l'extraction pourrait générer un programme incorrect sans que l'on puisse s'en apercevoir, sauf en relisant ou en exécutant le programme fautif.

Certes, l'extraction décrite partie 2 est suffisamment simple pour qu'un humain puisse être convaincu de la correction d'un programme OCaml l'implémentant. Mais qu'en est-il de l'extraction implémentée en pratique ? Cette extraction réalise beaucoup de simplifications sur le code extrait à tel point que le programme résultant peut différer considérablement du terme Coq source. Il devient alors légitime d'exiger une preuve que toutes ces transformations préservent la sémantique du programme Coq. C'est ce que nous proposons dans cette partie.

Ce travail est motivé par l'utilisation dans plusieurs contextes de l'extraction de Coq. En particulier, le compilateur certifié [11] d'un C réduit vers de l'assembleur PowerPC développé en Coq est un exemple marquant. Un compilateur de ML en Coq est en cours de développement [5].

Nous avons emprunté à Z. Dargaye sa formalisation en Coq d'un mini-ML et de sa sémantique et nous en sommes servi pour implémenter une nouvelle extraction certifiante. Ainsi, notre extraction générera des programmes mini-ML directement compilables par le compilateur certifié, et les preuves de correction générées pourront se combiner avec les preuves de correction du compilateur.

Cette extraction, appelée *extraction interne*, en est encore à un stade embryonnaire. Actuellement, c'est exactement l'extraction décrite section 2 qui est implémentée.

3.2 Mise en œuvre de l'extraction interne

Notre but est d'implanter dans Coq une extraction qui génère aussi une preuve que le terme extrait s'évalue bien « de la même façon » que le terme Coq d'origine. Lors du stage, nous avons travaillé avec des fonctions simples sur les entiers telles que l'addition ou le prédécesseur, ayant une partie logique limitée voire inexistante.

3.2.1 Langage cible

Nous souhaitons produire des objets Coq afin de pouvoir énoncer des résultats sémantiques dessus. Cependant, les termes extraits ne sont pas forcément directement exprimables en Coq. De plus, si nous souhaitons combiner notre extraction avec un compilateur de mini-ML certifié, nous devons d'une manière ou d'une autre exprimer précisément en Coq la sémantique de notre

langage. Ainsi, nous devons passer par un codage précis de notre langage cible. Notre fonction d'extraction génère des objets Coq du type suivant :

Définition 3.1 (type `term`)**.** Le type des *termes* mini-ML est défini en Coq par :

```
Inductive term : Set :=
  | TDummy : term
  | TVar : nat -> term
  | TLet : term -> term -> term
  | TFun : term -> term
  | TFix : term -> term
  | TApply : term -> term -> term
    (* "TApply t u" sera noté "t @ u" *)
  | TConstr : nat -> list term -> term
  | TMatch : term -> list pat -> term
with pat : Set :=
  | Patc : nat -> term -> pat.
```

La construction `TFun` lie une variable — l'argument — alors que `TFix` en lie deux : ♮0 (ou `TVar 0`) est l'argument, et ♮1 est la fonction pour l'appel récursif. Les constructeurs sont identifiés par un numéro, qui sera l'indice de la bonne branche dans la liste de patterns d'un `TMatch`, et le nombre dans un pattern indique le nombre de variables liées par ce pattern.

Si `f` est le nom d'une constante définie, alors :

```
Internal Extraction f.
```

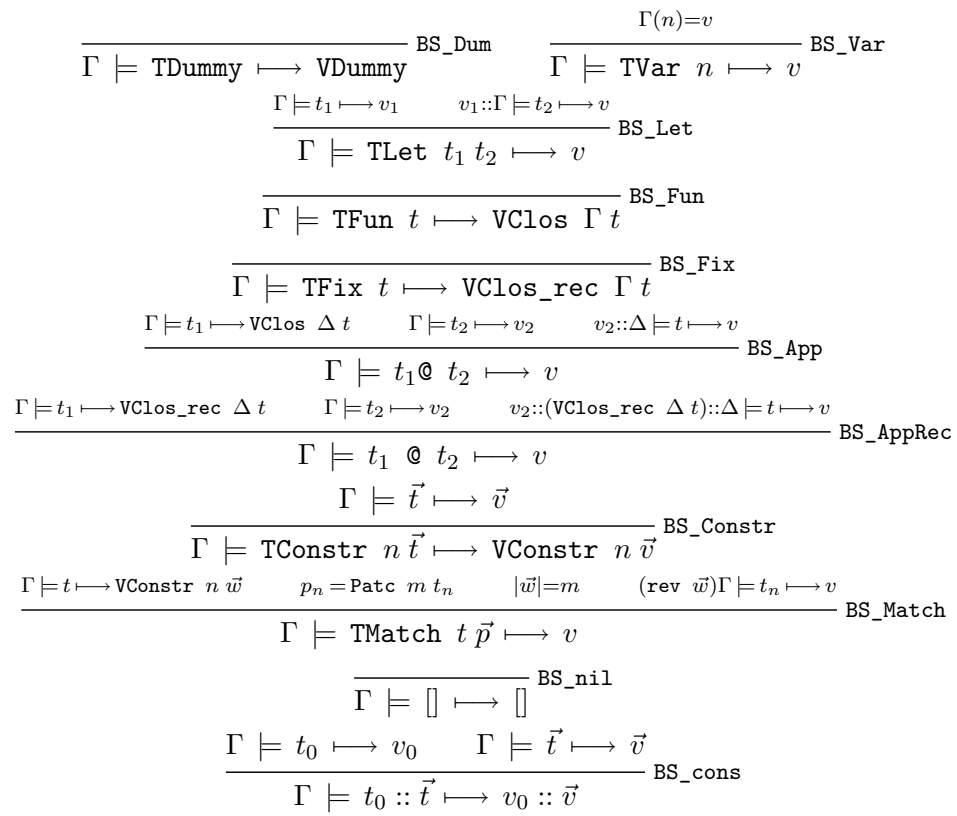
définit un objet `f__extr` de type `term` correspondant à l'extraction de `f`.

Pour la sémantique, nous aurons aussi besoin d'un type de terme évalué.

Définition 3.2 (type `value`)**.** Le type des *valeurs* mini-ML est défini par :

```
Inductive value : Set:=
  | VDummy : value
  | VClos : list value -> term -> value
  | VClos_rec : list value -> term -> value
  | VConstr : nat -> list value -> value.
```

Ces types sont très proches de ceux utilisés par Z. Dargaye dans la formalisation du langage mini-ML mentionnée précédemment. La seule différence notable est l'apparition de la constante `TDummy` (et de son pendant `VDummy`), qui représente tout simplement le □ de la section 2. Nous avons en effet souhaité garder une trace explicite des anciens termes logiques. Cependant, dans nos développements actuels, cette constante ne joue aucun rôle : en particulier, les règles `dummy_iota` et `dummy_fix` de la partie 2 n'ont pas



TAB. 7 – Sémantique à grands pas avec environnements – `BigStep(_list)`

d'équivalent. Ces règles ne sont nécessaires que pour des cas très particuliers que nous n'avons pas encore considérés. À terme, `TDummy` est destiné à être implanté par d'autres constructions mini-ML.

Nous avons également utilisé une sémantique à grands pas avec environnements fournie par Z. Dargaye :

Définition 3.3 (environnement, prédicats `BigStep`, `BigStep_list`)**.** Un *environnement* est une liste de valeurs. Les sémantiques `BigStep` (d'un terme) et `BigStep_list` (d'une liste de termes) sont données table 7.

Nous aimerions ainsi pouvoir générer et prouver automatiquement des théorèmes de la forme :

$$\forall n : \texttt{nat}, \quad [] \models \texttt{f\_\_extr}\ \texttt{@}\ n \longmapsto f\ n \tag{1}$$

lorsque f est un fonction de type `nat -> nat` et `f__extr` son extraction (de type `term`). Le lecteur attentif remarquera que l'expression ci-dessus n'est pas bien typée a priori. En réalité, elle fait appel au mécanisme de

coercions de Coq : des fonctions de coercions (encore appelées fonctions d'*internalisation*) de `nat` vers `term` et de `nat` vers `value` ont été définies, et elles sont automatiquement appelées lorsque nécessaire. Ainsi, `0` donne `TConstr 0 nil`, et `1` donne `TConstr 1 (TConstr 0 nil :: nil)`. Cela permet d'alléger considérablement la présentation des théorèmes de correction. Néanmoins, en interne, les coercions sont toujours explicitées. Remarquons aussi que nous n'avons pas besoin de formaliser la sémantique de Coq : on utilise directement la réduction de Coq pour évaluer l'expression $f\ x$.

3.2.2 Deux exemples

Nous allons donner deux exemples significatifs de ce sur quoi nous nous sommes concentrés.

Addition Il s'agit là d'une fonction récursive ne contenant aucun contenu logique.

```
Fixpoint plus (n m : nat) {struct n} : nat :=
  match n with
  | 0 => m
  | S p => S (plus p m)
  end
```

Nous donnons à titre d'illustration le terme extrait généré :

```
TFix
  (TFun
     (TMatch (TVar 1)
        (Patc 0 (TVar 0)
         :: Patc 1 (TConstr 1
                      (TVar 3 @ TVar 0 @ TVar 1 :: nil))
         :: nil)))
```

Prédécesseur avec précondition et postcondition Nous illustrons ici la définition via le mode preuve d'une fonction pleinement spécifiée :

```
Definition pred : forall n, n <> 0 -> {p | n = S p}.
Proof.
  intros; destruct n.
  destruct H; simpl; auto.
  exists n; auto.
Defined.
```

Cette fonction possède des parties logiques qui seront éliminées par l'extraction. On remarquera que cette fonction n'est pas récursive. Le terme Coq correspondant est :

```
fun (n : nat) (H : n <> 0) =>
match n as n0 return (n0 <> 0 -> {p : nat | n0 = S p}) with
| 0 =>
    fun H0 : 0 <> 0 =>
    match H0 (refl_equal 0) return {p : nat | 0 = S p} with
    end
| S n0 =>
    fun _ : S n0 <> 0 =>
    exist (fun p : nat => S n0 = S p) n0 (refl_equal (S n0))
end H
```

et le terme extrait est :

```
TFun
  (TFun
     (TMatch (TVar 1)
        (Patc 0 (TFun (TMatch TDummy nil))
         :: Patc 1
              (TFun (TConstr 0 (TDummy :: TDummy
                                :: TVar 1 :: TDummy
                                :: nil)))
            :: nil) @ TDummy))
```

3.3 Vers une sémantique plus adaptée ?

Les preuves directement exprimées dans la sémantique `BigStep` se sont avérées être plutôt pénibles, aussi avons-nous cherché d'autres présentations qui rendraient la recherche automatique de preuves plus facile.

Nous avons remarqué plus haut que nous pouvions utiliser Coq lui-même pour évaluer une expression comme $f\ x$ dans les théorèmes de correction. Poussons le concept plus loin en confiant la gestion de l'environnement dans la sémantique de mini-ML à Coq. Cela est réalisé en utilisant systématiquement des substitutions plutôt que des environnements, comme dans la sémantique `redn_term` de la table 4. La présentation avec environnement est pertinente pour réaliser des évaluateurs efficaces, mais celle avec substitution semble plus adaptée aux preuves.

Sur le même modèle que `redn_term`, une sémantique à petits pas avec substitutions (nommée `SmallStep`) a été définie pour mini-ML. Elle ressemble beaucoup à `dummy_redn_term`, aussi ne la détaillerons-nous pas ici. Nous noterons tout simplement $t \rightarrow u$ une étape de réduction `SmallStep`. P. Letouzey a prouvé son équivalence avec `BigStep`. En fait, nous utiliserons surtout la clôture réflexive transitive de `SmallStep`.

Il est à noter qu'avec les substitutions, les valeurs sont des cas particuliers de termes et il n'y a plus lieu de parler du type `value`. Un prédicat `IsValue` sur les `term` est plutôt utilisé.

3.4 Des briques pour les preuves de correction

La preuve des théorèmes de correction consiste essentiellement à mettre en parallèle la sémantique de mini-ML et l'évaluation du terme Coq correspondant : pour chaque étape d'évaluation du terme Coq, il faut deviner la règle de réduction mini-ML correspondante. Cela est possible car les termes extraits suivent fidèlement la forme des termes d'origine. Comme souvent en informatique, cela peut se faire facilement — voire automatiquement — pour peu que l'on adopte une bonne formulation.

Nous cherchons à prouver les théorèmes de correction tels que (1). Plutôt que de construire directement un terme du CCI en tant que preuve, nous souhaitons bénéficier du système Coq qui permet de construire incrémentalement — et de façon très naturelle — une preuve.

En Coq, les preuves sont dirigées par le but. Pour chaque type de conclusion, nous devons donc avoir une proposition qui permette d'y aboutir. Et c'est encore mieux s'il n'y en a qu'une seule. Nous allons dans la suite de cette section détailler trois cas nécessaires à l'établissement des preuves de correction de nos exemples `plus__extr` et `pred__extr`.

3.4.1 Réduction d'une application

Dans la table 7, nous constatons que les règles `BS_App` (la *β-réduction*) et `BS_AppRec` (la *ι-réduction* des points fixes) sont très similaires ; en particulier, leurs buts ont la même forme. Cette présentation n'est pas très pratique lorsqu'il faut « deviner » la règle à appliquer. En effet, la règle à appliquer n'est pas forcément immédiate : dans `f @ x`, il faut évaluer (au sens mini-ML) `f` afin de déterminer la bonne règle. Cette évaluation n'est a priori pas facile à réaliser : il faudrait passer par un lemme intermédiaire déclarant l'existence d'un terme `t` tel que `f` se réduit vers `TFun t` ou `TFix t` afin de pouvoir continuer la preuve pour `f @ x`. En fait, une fois le terme `t` trouvé, on peut directement le « sortir du chapeau » dans la preuve concernant `f @ x`, et ainsi appliquer la bonne règle.

Heureusement, Coq dispose d'un mécanisme pour rendre naturel ce genre de raisonnement : les *variables existentielles*. Il s'agit de « trous » que l'on peut placer au milieu d'une preuve, et qui sont destinés à être renseignés plus tard. Par exemple, dans le cas de `f @ x`, on peut créer une variable existentielle `?200`, puis on prouve que `f` se réduit vers `?200`. La preuve de réduction de `f` peut alors s'inscrire tout naturellement dans celle de `f @ x`. Au final, `?200` finira par être instanciée, et on pourra continuer la preuve pour `f @ x`. Les variables existentielles sont mise en œuvre notamment avec les tactiques `eapply` et `eauto`.

Pour pouvoir exploiter ainsi les variables existentielles, la formulation précise des lemmes utilisée est cruciale. Nous devons disposer d'un ensemble de propositions dont les conclusions couvrent tous les cas exactement une

seule fois, sinon la mauvaise règle risque d'être appliquée et l'utilisateur averti de Coq sait que les preuves impliquant des existentiels parties dans de mauvaises directions peuvent faire perdre beaucoup de temps.

Nous devons donc réunir d'une manière ou d'une autre β-réduction et ι-réduction des points fixes. Nous avons choisi d'introduire le prédicat suivant :

```
Definition SmallSteps_fun t1 v1 v :=
  match t1 with
    | TFun t => (t[0 := v1] ==> v)
    | TFix t => (t[0 := v1][0 := t1] ==> v)
    | _ => False
  end.
```

Ici, `==>` est une notation pour la clôture réflexive transitive $\rightarrow^*$ de $\rightarrow$. Ce prédicat permet d'énoncer une de nos briques de base pour prouver (1) :

```
Lemma SmallSteps_beta_iotafix : forall t1 t2 v t v1,
  (t1 ==> t) -> (t2 ==> v1) ->
  SmallSteps_fun t v1 v ->
  IsValue v1 ->
  (t1 @ t2 ==> v).
```

L'application de ce lemme via `eapply` va faire exactement ce que l'on a décrit plus haut. L'ordre des hypothèses fait que `t1 ==> t` va être prouvé dans un premier temps, et cette preuve va expliciter `t` qui pourra être utilisé par la suite pour réduire le prédicat `SmallSteps_fun t v1 v`[9].

3.4.2 Réduction des filtrages

La règle `BS_Match`, qui exprime la règle générale de *ι-réduction des filtrages*, ne pose pas de problème particulier et est agréable à utiliser sous la forme suivante :

```
Lemma SmallSteps_iota : forall t pl v n tl u,
  (t ==> TConstr n tl) ->
  nth_error pl n = Some (Patc (length tl) u) ->
  (u[0 ;;= rev tl] ==> v) ->
  IsValue_list tl ->
  clos_list tl ->
  (TMatch t pl ==> v).
```

[9]Ce raisonnement nous a permis de mettre en évidence un bug dans le système de tactiques (ou dans sa documentation). Il a été corrigé partiellement dans la révision 9985.

3.4.3 Réduction sous les constructeurs

La règle `BS_Constr` est en fait une règle de passage au contexte, cas particulier de `(W)Ctx_cst`. Nous avons adopté la formulation suivante pour cette règle :

```
Lemma SmallSteps_constr : forall n tl vl,
  SmallSteps_list tl vl -> (TConstr n tl ==> TConstr n vl).
```

où `SmallSteps_list` est le prédicat inductif suivant :

```
Inductive SmallSteps_list : list term -> list term -> Prop :=
  | SmallSteps_list_nil : SmallSteps_list nil nil
  | SmallSteps_list_const : forall t tl v vl,
      (t ==> v) ->
      SmallSteps_list tl vl ->
      SmallSteps_list (t::tl) (v::vl).
```

3.5 Preuves par induction

La règle de ι-réduction des points fixes de mini-ML réduit inconditionnellement les points fixes, alors que Coq attend que l'argument de décroissance d'un point fixe soit spécifié avant de le réduire ; de plus, cet argument doit commencer par un constructeur. On rencontre là notre premier problème lié à la différence entre la réduction forte de Coq et la réduction faible de mini-ML. Comment gérer alors les fonctions récursives telles que `plus` ?

Dans le cas de cette fonction particulièrement simple, le problème est résolu presque accidentellement. En effet, lorsque l'on se retrouve face à un énoncé de la forme `TMatch t pl ==> v`, `v` est un terme Coq de la forme `match t with ...` . Il est alors naturel dans ce genre de cas de faire une analyse par cas sur `t`. Il se trouve qu'en raisonnant plutôt par induction sur `t`, on obtient — en tout cas pour des fonctions suffisamment simples telles que `plus` — *la* bonne hypothèse qui permettra de prouver la réduction des appels récursifs. Remarquez que l'induction fournit aussi en quelque sorte le constructeur nécessaire à l'appel récursif. Cependant, cette méthode n'est pas très satisfaisante et ne permet pas de traiter des fonctions récursives un peu plus complexes telles que la fonction d'Ackermann.

Nous avons finalement réalisé qu'une fonctionnalité présente dans Coq — `Functional Scheme` et la tactique associée `functional induction` — semble être beaucoup plus prometteuse. Cependant, ces « schémas fonctionnels » ne semblent pas fonctionner avec des types dépendants, qu'utilise `pred`. Du travail est encore à réaliser dans cette voie, mais nous ne l'avons pas plus explorée lors de ce stage.

3.6 La tactique `extauto`

Les paragraphes précédents esquissent une méthode de preuve systématique, qui a été implémentée sous la forme d'une tactique `Ltac`, et qui prouvent complètement les théorèmes de correction pour `plus` et `pred`.

4 Conclusion, perspectives

Nous avons présenté deux manières très différentes d'aborder la garantie formelle pour l'extraction Coq : la certification de la fonction d'extraction elle-même dans le cadre de Coq-en-Coq, et la génération de preuves de correction des programmes extraits via une extraction interne dans le « vrai » Coq. Les résultats dans les deux cas paraissent prometteurs.

Cependant, nous n'avons traité qu'une extraction très simple, produisant des termes généralement verbeux et ne gérant pas les constantes. L'optimisation des programmes extraits — en éliminant *complètement* les □ ou `TDummy` inutiles — n'a pas été abordée.

Dans la section 3, la construction `let ... in ...` a été présentée, mais n'a pas été abordée dans nos exemples. Cette construction est liée à la gestion des constantes et à la manière de combiner plusieurs preuves de correction. De plus, nous avons évoqué l'utilisation de `functional induction` pour prouver la correction des fonctions récursives, mais sans creuser plus. Nous n'avons pas nous plus parlé ici de la génération des *énoncés* des théorèmes de correction, ainsi que des fonctions de coercions qui sont implicitement utilisées, même si nous pensons que cette génération ne devrait pas être très difficile.

Notre approche de l'extraction interne est très pragmatique : nous n'abordons que la correction de fonctions totalement appliquées. Nous ne traitons essentiellement que le type `nat` : passer à d'autres types de données simples ne devrait pas poser de problème, mais pour passer à tout type (y compris ceux contenant des sous-types fonctionnels), il faudra prouver la correction de n'importe quelle expression. Pour cela, on peut faire appel à la réalisabilité, comme dans l'étude sémantique de [12].

Enfin, nous avons occulté l'exécution par de vraies machines de nos programmes extraits. C'est le rôle des interpréteurs et des compilateurs. Notre but ultime est la compilation en programme machine. Cette phase est complexe, en particulier pour les langages fonctionnels, qui offrent un très haut niveau d'abstraction. Le développement d'un compilateur de mini-ML certifié est en cours [5], ce qui, combiné à nos travaux, permettra d'avoir une chaîne complète de certification depuis la spécification d'une fonction en Coq jusqu'à son exécution sur un processeur PowerPC [11].

Références

[1] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

[2] 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.

[3] S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[4] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[5] Z. Dargaye. Décurryfication certifiée. In *Journées Françaises sur les Langages Applicatifs JFLA'07*, 2007.

[6] 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.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580,583, 1969.

[8] 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.

[9] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952.

[10] C. Kreitz. *The Nuprl Proof Development System, Version 5*. Cornell University, Ithaca, NY, 2002. Available at `http://www.nuprl.org`.

[11] X. Leroy. Formal certification of a compiler back-end or : programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006.

[12] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[13] 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.

[14] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, January 1989.

[15] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15 :607–640, 1993.