Extraction certifiée dans Coq-en-Coq

Stéphane Glondu

Laboratoire Preuves, Programmes et Systèmes,
Université Paris Diderot - Paris 7,
Case 7014, 75205 Paris CEDEX 13, France
Stephane.Glondu@pps.jussieu.fr

Résumé

L'assistant de preuves Coq permet la génération de programmes corrects par construction. Cette fonctionnalité, appelée extraction, est exploitée notamment pour produire des bibliothèques de fonctions certifiées. Nous présentons dans cet article une formalisation de l'extraction en Coq, ainsi que certaines de ses propriétés qui ont été prouvées formellement. Ces travaux s'inscrivent dans le cadre de la formalisation de Coq en Coq de B. Barras.

1. Introduction

Programmation fonctionnelle et λ-calcul

Les langages fonctionnels — tels qu'OCaml, Haskell ou Scheme — sont des langages dérivés du λ-calcul. 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 programmes et preuves. 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 programmes et des systèmes informatiques en tous genres. Ils sont parfois utilisés 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 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 dans ce formalisme, toutes les preuves *sont* des programmes. Cependant, dans ces programmes, on peut distinguer des parties purement logiques et des parties vraiment calculatoires, distinction 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, `Prop`. 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. Après effacement de ces parties logiques, il est souhaitable de garder une relation entre le programme extrait et la proposition de départ, afin de pouvoir bénéficier de ce qui a été prouvé. À cette fin, on peut utiliser la notion de réalisabilité, 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 défini une relation 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, lorsqu'on travaille avec des assistants de preuves, la tentation est grande de prouver de tels résultats en Coq. L'objectif des travaux présentés ici 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].

Contribution

Nous présentons ici une extraction formalisée 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 site web de l'auteur. La partie concernant l'extraction proprement dite fait environ 1 200 lignes de Coq et a demandé quatre mois de développement.

Nous présentons brièvement, dans la section 2, la formalisation de Coq de B. Barras, puis l'extraction proprement dite dans la section section 3. 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

2.1. Les Systèmes de Types Purs

B. 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 :

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

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 lorsqu'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 pair 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 :

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

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.

En outre, nous noterons $\uparrow_k^n t$ le terme t où toutes les variables libres sous k lieurs de t sont incrémentées de n. Cette opération est appelée *relocation*.

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 :

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

Ici, $\mathcal{P}(\texttt{X})$ désigne l'ensemble des parties de `X`, ce que l'on peut écrire `X -> Prop` en Coq.

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 les suivantes :

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

$$\frac{\Gamma \vdash \quad (s_1, s_2) \in \texttt{axiom}}{\Gamma \vdash s_1 : s_2}\ \texttt{Typ\_srt} \qquad \frac{\Gamma \vdash \quad \Gamma(n) = \delta}{\Gamma \vdash \natural n : \uparrow_0^{n+1} T_\delta}\ \texttt{Typ\_rel} \qquad \frac{\Gamma \vdash \quad [\_ : T] \vdash \quad (c, T) \in \Sigma_0}{\Gamma \vdash c : T}\ \texttt{Typ\_cst0}$$

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

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

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

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

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

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 cet article 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. Les règles suivantes définissent la *clôture par contexte* $\rightarrow_{\mathcal{X}}$ de $\rightarrow$:

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

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

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

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

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

Si dans un système, la réduction est close par contexte, elle sera qualifiée de *forte*.

Les `(W)` apparaissant dans la définition ci-dessus seront expliqués plus tard, lors de la définition 3.2.

2.2. Le Calcul des Constructions Inductives

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

2.2.1. Sortes

Nous disposons d'une hiérarchie de sortes comprenant deux sortes imprédicatives `Prop` et `Set`[1], 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[2].

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

Marques Les *marques* jouent un rôle important dans le typage des points fixes que nous ne détaillerons pas ici. Il servent également 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 (prenant une paire en argument).

[1] Cette hiérarchie de sortes correspond à celle de Coq 6.2. L'imprédicativité de `Set` étant incompatible avec certains axiomes raisonnables [7], elle a été retirée à partir de la version 8.0 : `Set` n'est plus qu'une sorte prédicative au même titre que les $\texttt{Type}_i$. Ce détail a une importance marginale dans le cadre de l'extraction.

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

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[3]. 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 Dans le CCI, les définitions de fonctions récursives (appelées points fixes) doivent explicitement mentionner un argument *structurellement* décroissant, afin de garantir leur bonne fondation. La vérification est faite syntaxiquement dans Coq. Dans le développement de B. Barras, le critère est sémantique, et intégré au typage (utilisant les marques), mais nous allons ici nous contenter de l'intuition que l'argument du point fixe doit décroître avec chaque appel récursif. Il est à noter qu'ici, 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.

Constantes globales L'opérateur $\texttt{Const}\,\{x\}$ désigne une constante dont la définition est dans l'environnement global Δ sous le nom x. Il s'agit d'un opérateur unaire pouvant prendre en argument un *paramètre*, qui peut être référencé par $\natural 0$ dans le corps de 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[4] est identifiée par I dans l'environnement global Δ. C'est dans Δ 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.

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

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

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 suivantes :

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

$$\frac{}{\pi_1(M,N) \rightarrow_{\pi_1} M}\ \texttt{proj1} \qquad \frac{}{\pi_2(M,N) \rightarrow_{\pi_2} N}\ \texttt{proj2} \qquad \frac{y = x_i}{\langle \vec{x} \doteq \vec{t} : \vec{T} \rangle_{\langle y \rangle} \rightarrow_{\pi_r} t_i}\ \texttt{projr}$$

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

$$\frac{t = \texttt{Constr}\,\{C\}\,(M)}{\texttt{Fix}(f : T)\ p\ t \rightarrow_{\iota_{\texttt{fix}}} f\ (\texttt{Fix}(f : T))\ p\ t}\ \texttt{iota\_fix} \qquad \frac{t = \texttt{Constr}\,\{C\}\,(M) \qquad p_i = C}{\texttt{Case}\,\{\vec{p}\}\,(t, P, \vec{f}) \rightarrow_{\iota_{\texttt{case}}} f_i\,M}\ \texttt{iota\_case}$$

On notera $\rightarrow_{\mathcal{C}}$ 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.

3. Extraction vers $\text{CCI}_{\square}$

3.1. Prédicat d'extraction partielle

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 3.1 (prédicat `Pe`)**.** La relation d'extraction $\Gamma \vdash t \rightarrow_{\mathcal{E}} t'$ est définie par les règles suivantes :

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : \texttt{Prop}}{\Gamma \vdash t \rightarrow_{\mathcal{E}} \square}\ \texttt{Pe\_Prop} \qquad \frac{t \in \texttt{sort}}{t \rightarrow_{\mathcal{E}} \square}\ \texttt{Pe\_Srt} \qquad \frac{t \in \texttt{oper}}{t \rightarrow_{\mathcal{E}} \square}\ \texttt{Pe\_Cst0}$$

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

$$\frac{c \in \{\texttt{Ind}\,\{I,n\}, \texttt{Record}\,\{\vec{x}\}\}}{c(t) \to_{\mathcal{E}} \square}\ \texttt{Pe\_\{MutInd,Record\}} \qquad \frac{t \to_{\mathcal{E}} t' \quad i \in \{1,2\}}{\pi_i(t) \to_{\mathcal{E}} \pi_i(t')}\ \texttt{Pe\_Proj}i$$

$$\frac{t_1 \to_{\mathcal{E}} t_1' \qquad t_2 \to_{\mathcal{E}} t_2'}{t_1\, t_2 \to_{\mathcal{E}} t_1'\, t_2'}\ \texttt{Pe\_App} \qquad \frac{t_1 \to_{\mathcal{E}} t_1' \qquad t_2 \to_{\mathcal{E}} t_2'}{(t_1, t_2) \to_{\mathcal{E}} (t_1', t_2')}\ \texttt{Pe\_Pair}$$

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

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

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

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

On dira alors que t s'*extrait* (partiellement) vers t' dans le contexte Γ.

Dans cette définition, on peut distinguer trois types de règles :
- `Pe_Prop` élimine les parties logiques ;
- les autres règles produisant $\square$ éliminent ce qui est lié au typage, comme les opérateurs sans argument (`Pe_Cst0`) ou les types eux-mêmes ;
- 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 : bien que la plupart des règles soient dirigées par la syntaxe, la règle `Pe_Prop` peut intervenir n'importe où. Un chevauchement est ainsi possible entre les règles.

Comme mentionné à la remarque 2.5, nous omettrons souvent le « $\Gamma \vdash$ ». Notons que si le terme t' reste syntaxiquement un terme de CCI, il appartient à un sous-langage plus proche du λ-calcul non typé que du CCI, langage que nous noterons $\text{CCI}_{\square}$. C'est ce qui jouera le rôle de notre ML. Nous avons choisi de conserver la même syntaxe afin d'alléger le développement. 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.

3.2. 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 ML. Contrairement au CCI, les sous-termes d'une abstraction ne sont pas réduits en $\text{CCI}_{\square}$, ce qui motive la définition suivante :

Définition 3.2 (opération `wctxt`)**.** Soit $\to$ une règle de réduction. La *clôture par contexte faible* $\to_{\mathcal{W}}$ est définie comme `ctxt` (définition 2.6), en enlevant les règles `Ctx_lam_r` et `Ctx_prd_r`.

De plus, si on regarde la définition 2.9, 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. 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 les deux règles de réduction `dummy_fix` et `dummy_case` :

Définition 3.3 (prédicats `dummy_redn_term` et $\rightarrow_\square$)**.** La règle `dummy_redn_term` est formée de l'union de `redn_term` (définition 2.9) et des règles suivantes :

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

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

On notera $\rightarrow_\square$ la clôture par contexte faible de `dummy_redn_term`.

Petit point sur les notations « $\Gamma \vdash t \rightarrow_\square u$ » 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 $\text{CCI}_\square$.

3.3. 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 dans le cadre du travail présenté ici. Les présentations des lemmes ci-dessous feront appel à des notions et notations qui ne seront pas définies formellement ici, mais juste expliquées informellement. Bien sûr, ces notions ont été formellement définies en Coq.

Lemme 3.4 (`Pe_weak`)**.** *La règle suivante est admissible :*

$$\frac{\Gamma_1\Gamma_2 \vdash t \rightarrow_\mathcal{E} t' \qquad \Gamma_1\delta \vdash}{\Gamma_1\delta\Gamma_2' \vdash \uparrow^1_{|\Gamma_2|} t \rightarrow_\mathcal{E} \uparrow^1_{|\Gamma_2|} t'}$$

Cette règle dit que l'extraction est stable par ajout d'une déclaration δ dans un environnement $\Gamma_1\Gamma_2$. Γ_2' représente l'environnement Γ_2 ayant subi les opérations idoines de relocation. On remarquera la ressemblance entre ce lemme et le lemme d'*affaiblissement* que l'on rencontre en typage[5]. 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*[6] :

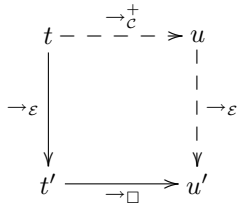
Lemme 3.5 (`Pe_sub`)**.** *La règle suivante est admissible :*

$$\frac{\Gamma_1[x:U] \vdash t:T \qquad \Gamma_2 \vdash u:U \qquad t \rightarrow_\mathcal{E} t' \qquad u \rightarrow_\mathcal{E} u'}{t\,\{\natural 0 \leftarrow u\} \rightarrow_\mathcal{E} t'\,\{\natural 0 \leftarrow u'\}}$$

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 3.6 (`Pe_wctxt_correct`)**.** *Soient t un terme CCI clos (sans variable libre, ni constante globale) bien typé et t', u' deux termes de $\text{CCI}_\square$ 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 $t \rightarrow_\mathcal{C}^+ u$.*

[5] lemme 5.25 `typ_weak` de [1], page 140
[6] lemme 5.26 `typ_sub` de [1], page 140



En Coq, ce théorème de relèvement s'exprime sous la forme suivante :

```
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 cependant plus longue 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 3.7 (`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 pas encore prouvé que le calcul ne bloque pas trop tôt, sur un terme non totalement réduit vers une valeur.

3.4. Une extraction extraite

B. Barras a réalisé un prototype de système de preuves dans lequel le typeur est obtenu via la « vraie » extraction de Coq, auquel est ajoutée une interface sommaire écrite en OCaml. Nous avons ajouté à ce système 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.

4. Conclusion, perspectives

Nous avons présenté une formalisation de l'extraction dans le cadre de Coq-en-Coq. 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 □ inutiles — n'a pas encore été abordée.

Cela n'est qu'une manière d'aborder la garantie formelle pour l'extraction Coq. D'autres travaux (en cours) visent à concevoir une extraction pour le « vrai » sytème Coq, générant des preuves de correction en même temps que les programmes extraits.

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] H. Geuvers. *Inconsistency of classical logic in type theory*. Short note, 2001.

[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.