Vers une certification de l'extraction de Coq

Stéphane Glondu

Université Paris Diderot (Paris 7)
École doctorale de
Sciences Mathématiques de
Paris Centre

Thèse

pour l'obtention du diplôme de

Docteur de l'Université Paris Diderot

spécialité Informatique

Vers une certification de l'extraction de Coq

présentée et soutenue publiquement par

Stéphane Glondu

le 1er juin 2012

devant le jury composé de

directeur	M.	Roberto Di Cosmo
co-directeur	M.	Pierre Letouzey
rapporteurs	M.	Stefano Berardi
	M.	Jean-François Monin
examinateurs	M.	Bruno Barras
	Mme	Christine Paulin-Mohring

Version 1b7c69f compilée le mardi 5 juin 2012, 15:44:06 (UTC+0200)

Remerciements

Je voudrais remercier tout particulièrement Pierre Letouzey pour l'encadrement de cette thèse. Son expertise et sa grande disponibilité m'ont apporté une aide précieuse et irremplaçable. Je tiens également à remercier Roberto Di Cosmo pour son soutien.

Je suis aussi très reconnaissant envers Stefano Berardi et Jean-François Monin pour avoir accepté d'être rapporteurs de ma thèse, avec tout le travail que cela implique.

Le travail de formalisation de Coq réalisé par Bruno Barras est formidable et a directement servi de base à une grande partie de mes travaux ; je suis honoré que Bruno ait accepté de m'évaluer. Je remercie également Christine Paulin, sans qui Coq ne serait pas ce qu'il est aujourd'hui, et qui a également accepté de faire partie du jury.

Ma gratitude va également envers tous ceux qui ont lu et commenté les premières versions de ce manuscrit, en particulier Véronique, Nicolas, Hugo et Édouard.

J'ai aussi eu l'occasion d'interagir avec beaucoup de personnes très compétentes lors de séminaires et de conférences, mais aussi dans le cadre du développement de Coq. Je ne vais pas me hasarder à les énumérer toutes, mais je les remercie pour toutes les discussions que j'ai pu avoir avec elles et qui ont pu m'inspirer dans mes travaux.

Je tiens aussi à remercier tous les membres du laboratoire PPS, qui m'a accueilli et fourni un environnement de travail stimulant, et en particulier tous les co-bureaux que j'ai eus au fil des années : Sylvain, Christine, Stéphane, Paolo, Grégoire, Séverine, Mehdi, Élie, Vincent, Danko, Pierre, Matthias et Nicolas… et je m'excuse auprès de ceux que j'aurais pu oublier ! Je tiens aussi à remercier Odile Ainardi, qui m'a apporté son aide tout au long de ma thèse.

Je voudrais aussi remercier tous les professeurs qui ont contribué à ma formation. Il est difficile de tous les nommer ici, mais je tiens à mentionner en particulier Hubert Comon, grâce à qui je me suis intéressé à l'informatique théorique.

Mes pensées vont aussi à mes parents, qui m'ont encouragé tout au long de mes études tout en me laissant une totale liberté. J'adresse aussi un grand merci à mes frères, au reste de ma famille et à mes amis pour leur support tout au long de mes études.

Table des matières

1 Introduction

1.1 Enjeux

Notre société repose de plus en plus sur les ordinateurs. La plupart des machines modernes sont contrôlées par un ordinateur, qui exécute méticuleusement des suites d'instructions élémentaires. De nos jours, ces instructions sont décrites sous une forme abstraite, les *programmes*, un peu à la manière de recettes de cuisine. Les informaticiens construisent du matériel toujours plus performant et petit, capable d'exécuter des instructions élémentaires toujours plus diverses et évoluées, mais construisent également du *logiciel*, c'est-à-dire de quoi contrôler le matériel, et en particulier les programmes, auxquels sont dévolues de plus en plus de tâches.

Isolément, la plupart des instructions élémentaires sont relativement faciles à comprendre. Mais les systèmes informatiques actuels peuvent en exécuter plusieurs milliards par seconde, ce qui rend ces systèmes tellement complexes que plus personne n'est capable de maîtriser seul tout ce qui s'y passe. Tous les jours, de nouvelles erreurs de programmation — ou *bugs* — sont découvertes, et d'autres sont corrigées. Quelle confiance accorder alors à l'informatique ?

C'est aussi le rôle des informaticiens d'assurer cette confiance, et ils ont développé à cette fin des procédures visant à l'améliorer : tests automatiques, audit par plusieurs personnes indépendantes... En 1969, C. A. R. Hoare [26] propose de *raisonner* sur les programmes avec la même rigeur que les mathématiciens : les méthodes formelles asseyent ainsi leur position dans la panoplie de l'informaticien, désormais capable de *certifier* qu'un programme respecte une spécification.

1.2 Logique et informatique

Les méthodes formelles reposent sur la *logique* — l'étude du raisonnement — qui a pris son essor au XX^{e} siècle, peu de temps avant l'informatique. Plusieurs logiciens ont proposé des formalismes permettant d'écrire des énoncés (ou *formules*) et de les prouver avec des briques élémentaires de raisonnement telles que le *modus ponens* :

$$\frac{A \qquad A \Rightarrow B}{B}$$

La règle ci-dessus doit se comprendre ainsi : « si A est vrai, et que A implique B, alors B est vrai ». En 1935, G. Gentzen propose un ensemble de règles sur ce modèle, la

déduction naturelle, qui a inspiré beaucoup d'autres systèmes par la suite.

Écrire des preuves complètes en déduction naturelle peut être fastidieux ; les vérifier l'est encore plus, mais est facilement mécanisable. La démocratisation des ordinateurs a été une aubaine pour la mise en pratique de systèmes logiques formels : de nos jours, ces systèmes sont au cœur de programmes — les *assistants de preuve* — capables de vérifier des preuves formelles et d'aider à leur élaboration et, parfois, de les générer automatiquement. Il existe aujourd'hui de nombreux assistants (Isabelle, Matita, PVS…) et prouveurs automatiques (Alt-Ergo, Yices, Z3…).

Dans cette thèse, nous étudions un style de programmation particulier, la *programmation fonctionnelle*. Fonctionnelle, car les programmes y sont décrits sous forme de fonctions. Les langages de programmation qui favorisent ce paradigme — tels qu'OCaml, Haskell et Scheme — sont dérivés du λ-calcul, proposé par A. Church [12] dans les années 1930. Dans ce langage abstrait, les programmes sont des *termes* qui évoluent selon des règles appelées règles de *réduction*.

Il existe trois constructions syntaxiques de base dans le λ-calcul : la variable, l'abstraction et l'application. Une variable est un nom, permettant de référencer symboliquement l'argument d'une fonction dans son corps. L'abstraction $\lambda x.t$ construit une fonction prenant un argument x et retournant le terme t. L'application $f\ u$ désigne la fonction f appliquée à l'argument u ; dans le cas où f est de la forme $\lambda x.t$, cette application se réduit en t dans lequel les occurrences de x ont été remplacées par u :

$$(\lambda x.t)\ u \to t\,\{x \leftarrow u\}$$

Cette règle fondamentale, la *β-réduction*, est combinée à d'autres règles permettant la réduction de sous-termes.

Pour faire du λ-calcul un vrai langage de programmation, il est d'usage d'ajouter des constantes primitives (nombres, caractères…). La syntaxe vue ci-dessus autorise alors des termes « absurdes », tels que l'application d'un nombre à un terme. Pour pallier ce problème, une notion de *type* a été introduite : par exemple, l'ensemble des fonctions acceptant un argument de type T_1 et renvoyant un terme de type T_2 est désigné par le type $T_1 \to T_2$. Trois règles de typage contraignent la formation des termes et constituent le *λ-calcul simplement typé* [4] :

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \qquad \frac{\Gamma \vdash f : T_1 \to T_2 \qquad \Gamma \vdash u : T_1}{\Gamma \vdash f\ u : T_2}$$

Ces règles mettent en jeu des jugements de la forme $\Gamma \vdash t : T$, où Γ désigne un environnement contenant la liste des variables valides avec leurs types, t un terme, et T son type. Par exemple, la deuxième règle dit que $\lambda x.t$ est de type $T_1 \to T_2$ à condition que t soit de type T_2 après avoir ajouté dans l'environnement une nouvelle variable x de type T_1.

Dès 1958, H. Curry remarque [15] une analogie entre le λ-calcul simplement typé [1] et un système logique formel connu, la logique propositionnelle intuitionniste, établissant ainsi une correspondance entre types et formules, et entre programmes et preuves. Cela se voit bien en effaçant les termes dans les trois règles précédentes :

$$\frac{T \in \Gamma}{\Gamma \vdash T} \qquad \frac{\Gamma, T_1 \vdash T_2}{\Gamma \vdash T_1 \to T_2} \qquad \frac{\Gamma \vdash T_1 \to T_2 \qquad \Gamma \vdash T_1}{\Gamma \vdash T_2}$$

En 1969, W. A. Howard étend cette correspondance à la logique du premier ordre en proposant un λ-calcul avec types dépendants [27]. Cette correspondance est depuis couramment connue sous le nom de *correspondance de Curry-Howard*.

La formalisation des systèmes logiques a donné naissance à de nombreux assistants de preuves. Avec la correspondance de Curry-Howard en tête, il est tentant de déterminer le contenu calculatoire des preuves, d'*extraire* des programmes à partir de preuves. Les programmes extraits sont alors naturellement exprimés dans des langages fonctionnels. Cette idée a été mise en pratique dans plusieurs systèmes tels que Nuprl [30], Isabelle [9, 10] ou Minlog [8]. Nous nous intéresserons plus particulièrement à Coq [46].

1.3 Certification de programmes

Les assistants de preuves permettent de prouver formellement des théorèmes mathématiques usuels, mais aussi des propriétés sur des programmes et des systèmes informatiques en tout genre. 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. Plusieurs approches existent à la certification de programmes.

On peut par exemple chercher à prouver *a posteriori* des propriétés sur des programmes existants, éventuellement annotés. Cela peut se faire en analysant le *code source* (forme originalement écrite par le programmeur), le *code machine* (version comprise par l'ordinateur) ou n'importe quelle version intermédiaire que la *compilation* (transformation de code source en code machine) peut produire. Certains systèmes, comme Why [19], permettent de prouver des propriétés quelconques et peuvent faire appel à des prouveurs automatiques externes ou des assistants de preuve ; d'autres, comme Astrée [14], ne traitent que quelques propriétés spécifiques mais sont totalement automatisés.

Une autre approche consiste à exprimer le comportement souhaité dans un formalisme plus abstrait que les langages de programmation usuels, puis d'en dériver automatiquement des programmes satisfiant *a priori* les propriétés voulues. Les programmes ainsi générés le sont dans des langages de programmation usuels, mais le raisonnement se fait

1. En réalité, Curry utilisait la logique combinatoire, mais cette dernière est équivalente au λ-calcul.

dans le formalisme de départ et/ou sur le processus de génération lui-même. C'est plutôt dans ce cadre que se place cette thèse.

1.4 L'extraction de Coq

Coq est bâti directement sur la correspondance de Curry-Howard : toutes les preuves sont représentées en interne comme des termes du *calcul des constructions inductives*, ou CIC (en anglais, *calculus of inductive constructions*), 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 informatives, distinction qui peut déjà être sentie dans certains énoncés : ainsi, dans une proposition existentielle telle que « $\exists x, P(x)$ », où $P(x)$ désigne la proposition « x est premier », on peut être 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 vraiment savoir pourquoi. En pratique, $P(x)$ va correspondre à du code mort.

L'extraction en Coq a pour rôle de séparer ces différentes composantes, d'effacer les parties logiques afin de ne laisser que les parties informatives des preuves. En Coq, cela est réalisé en mettant les propositions purement logiques dans un type particulier, `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 sur le terme de départ. À cette fin, on peut utiliser la notion de réalisabilité, introduite par S. C. Kleene [29] en 1945. Notons que l'extraction n'est pas la seule raison d'être de `Prop` ; il possède également des propriétés logiques qui le rendent utile même en l'absence de l'extraction.

L'extraction en Coq a commencé avec la thèse de C. Paulin [36, 37, 39], qui a défini une relation de réalisabilité adaptée au calcul des constructions (CC), à la base du Coq de l'époque, et une extraction associée. Cette extraction théorique a été prouvée [37] et 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* (CIC), ont mené P. Letouzey à proposer une nouvelle extraction pour Coq dans sa thèse [33]. Cette extraction a été implantée dans Coq, et c'est elle qui est actuellement utilisée.

Une application remarquable de l'extraction est la certification dans le cadre du projet CompCert d'un compilateur réaliste pour le langage C [31] : les phases de transformation ont été programmées en Coq, puis extraites vers OCaml et interfacées avec du code d'entrée-sortie écrit manuellement afin d'obtenir un compilateur complet, transformant un fichier C en un programme exécutable, et ce pour plusieurs architectures réelles. La correction de ce compilateur repose en particulier sur la correction de l'extraction : il est supposé implicitement que les programmes générés par l'extraction se comportent

comme ceux écrits en Coq, sur lesquels les preuves ont été réalisées. Toujours dans le cadre de CompCert, on peut également citer le développement par Z. Dargaye d'un compilateur de mini-ML certifié [16].

Une autre application importante de l'extraction se trouve dans les travaux de B. Barras [5], qui a formalisé en Coq le typage d'une variante de CIC comparable à Coq. Le théorème de décidabilité du typage a été extrait vers OCaml, puis intégré au sein d'un assistant de preuve autonome. Ce nouveau système, très rudimentaire, ne dispose pas de toutes les facilités de Coq, mais il s'agit néanmoins d'une preuve de concept admirable.

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 cette thèse est de voir dans quelle mesure ces résultats peuvent être transposés en Coq.

Nos travaux s'insèrent entre ceux de B. Barras et de Z. Dargaye, et constituent une étape cruciale dans la validation de toute une chaîne de compilation de Coq vers du code machine.

1.5 Contributions

Cette thèse s'articule autour de deux grands axes : l'extraction certifiée et l'extraction interne.

Extraction certifiée Nous avons formalisé l'extraction dans le cadre du système de B. Barras [5], que nous avons dû adapter. Notre extraction travaille sur les termes d'une variante de CIC formalisée entièrement en Coq, et consiste à remplacer tous les sous-termes logiques par une constante inerte. C'est la première étape de l'extraction actuellement implantée dans Coq. Nous avons prouvé la préservation de la sémantique : les termes extraits se comportent de la même manière que les termes originaux, c'est-à-dire que les parties logiques correspondent bien à du code mort. Plus précisément, nous avons montré en Coq que les réductions dans le langage cible peuvent être mises en correspondance avec des réductions dans le langage source, et réciproquement, comme l'avait fait auparavant sur papier P. Letouzey [33]. Ces travaux ont été commencés en master [22] et ont partiellement fait l'objet d'une publication [23].

Même si nos travaux ne concernent pas directement l'extraction de Coq, plus évoluée, nous avons identifié une erreur significative dans son implantation, due à une divergence entre le CIC étudié théoriquement et celui implanté dans Coq, ce qui motive d'autant plus le rapprochement de la théorie et de la pratique.

Nous avons ajouté une commande d'extraction au système autonome de B. Barras, reposant sur l'extrait [2] de la formalisation. Nous proposons également d'utiliser la for-

2. Dans le langage courant, le mot *extraction* désigne le programme produit par l'extraction, mais dans cette thèse, il désigne plutôt le processus d'extraction lui-même. Nous utiliserons parfois le terme

malisation extraite directement au sein de Coq à travers un greffon (*plug-in*).

Les travaux autour de l'extraction certifiée ont été réalisés à un niveau très abstrait : le langage source n'est pas exactement le CIC implanté dans Coq, et le langage cible s'éloigne encore d'un environnement d'exécution concret. Nous avons également essayé une approche plus pragmatique, directement intégrée au système Coq existant, et visant un langage pour lequel il existe un compilateur certifié.

Extraction interne Indépendamment de ce CIC entièrement formalisé en Coq, nous avons étudié la possibilité de vérifier *a posteriori* les programmes extraits à partir de termes Coq authentiques. Pour cela, nous avons écrit une autre variante de l'extraction qui cible un mini-ML totalement formalisé dans Coq, mais sans aucune formalisation de Coq lui-même. Nous avons défini une notion de correction des programmes extraits, et nous avons développé des techniques dans le but de prouver facilement, voire automatiquement, cette correction.

Développements Les développements associés à l'extraction certifiée et à l'extraction interne sont disponibles en ligne [3] ou à la demande.

Perspectives Nous proposons également informellement une version de CIC revisitée pour être plus proche de Coq, dans l'optique d'une formalisation, puis d'une traduction automatique de développements existants. Ces idées peuvent faire l'objet de travaux futurs.

1.6 Plan

Nous commençons par une introduction au système Coq (chapitre 2), où nous présentons informellement CIC via des exemples, ainsi que des fonctionnalités annexes offertes par le système qui le rendent plus agréable pour l'utilisateur. Le chapitre 3 se concentre sur l'extraction proprement dite : nous y présentons une variante de CIC entièrement formalisée en Coq, $\mathrm{CIC}_{\mathrm{B}}$, ainsi que différents résultats de correction concernant l'effacement de parties logiques. Le chapitre 4 présente l'extraction interne, où nous considérons directement ce qui est implanté dans Coq comme langage source, sans le formaliser, et un mini-ML formalisé comme langage cible. Enfin, nous exposons quelques idées pour une formalisation (future) plus fidèle à Coq (chapitre 5).

extrait pour parler du résultat de l'extraction.

3. `http://stephane.glondu.net/`

2 Le système Coq

Coq [46] est un assistant de preuve interactif où les formules et les preuves sont représentées en interne par des termes d'un langage purement fonctionnel statiquement typé, le *calcul des constructions inductives*, ou CIC (en anglais, *calculus of inductive constructions*). Coq est conçu selon le principe de de Bruijn [48] : au cœur du système se trouve un vérificateur de types de CIC, appelé *noyau*, qui se veut aussi simple et fiable que possible. Mais interagir directement avec le noyau peut s'avérer pénible pour l'utilisateur final. Aussi Coq fournit-il d'autres fonctionnalités (inférence de types, notations, tactiques, automatisation, etc.) pour assister l'utilisateur. Ces fonctionnalités ne requièrent pas la même fiabilité que le noyau et ne risquent pas d'introduire d'incohérence logique dans le système, car elles reposent sur le noyau pour vérifier leurs propres résultats.

CIC est lui-même conçu de sorte que le noyau soit conceptuellement simple à implanter, même si en pratique l'implantation, soucieuse notamment des performances, est plus compliquée. La méta-théorie de CIC a été étudiée dans [47], et de nombreuses fonctionnalités ont pu être ajoutées à Coq sans pour autant modifier CIC, comme par exemple dans [7, 44, 24]. Une autre implantation de CIC existe au sein de l'assistant de preuve Matita [1].

La façon la plus élémentaire d'utiliser Coq est de soumettre des *commandes vernaculaires* à sa boucle d'interaction, `coqtop` :

```
Welcome to Coq 8.3pl2 (July 2011)

Coq < Check 18.
18
     : nat

Coq <
```

La première ligne est un message d'accueil affiché une seule fois par le système au démarrage. La boucle invite (`Coq <`) ensuite l'utilisateur à soumettre ses commandes. Dans cette session, la commande `Check` est invoquée. Elle prend en argument un terme, le vérifie, et renvoie son type. Nous avons reproduit ici une session texte, mais il existe d'autres interfaces pour interagir avec Coq [2]. L'utilisateur peut également réunir un ensemble de commandes dans un fichier (généralement avec l'extension `.v`) pour le soumettre di-

rectement d'un bloc à la boucle d'interaction, ou le compiler en un fichier `.vo` avec `coqc`, utilisable par la boucle d'interaction ou la compilation d'autres fichiers.

2.1 Noyau

On nomme traditionnellement CIC le langage implanté au cœur de Coq. Sa définition exacte est complexe, et dépend notamment de la version du système considéré. Nous n'en fournirons pas une description précise, mais nous en présentons ici quelques aspects, qui peuvent aider à mieux comprendre sa version formalisée dans le chapitre 3, $\mathrm{CIC_B}$. CIC est décrit plus en détail dans le manuel de référence de Coq [46].

2.1.1 Le calcul des constructions (CC)

CIC peut être vu en première approximation comme une extension du λ-calcul simplement typé [4] à la Church : les constructions de base pour le calcul sont la variable, l'abstraction (construction d'une fonction) et l'application. Par exemple, le traditionnel opérateur S de la logique combinatoire [15] peut s'écrire concrètement en Coq comme suit (en supposant que `A`, `B` et `C` désignent des types) :

```
fun (x : A -> B -> C) =>
  fun (y : B -> C) =>
    fun (z : A) => x z (y z)
```

ou, de façon plus concise :

```
fun (x : A -> B -> C) (y : B -> C) (z : A) => x z (y z)
```

Comme en OCaml, lorsqu'une variable n'est pas utilisée, un tiret bas (`_`) peut lui être substitué dans le lieur, et des commentaires peuvent être insérés entre `(*` et `*)` :

```
fun (x : A) (_ : B) => x (* opérateur K *)
```

Ces expressions font apparaître des types qui, contrairement au λ-calcul simplement typé, ne font pas partie d'une classe syntaxique dédiée. Au lieu de cela, la syntaxe des termes est étendue avec de nouvelles constructions permettant de représenter les types : ainsi, le type de `fun (x:A) => t` est noté `forall (x:A), B` si `t` est de type `B`. Dans cet exemple, la variable `x` peut apparaître libre dans `t` et `B`. Le cas où `x` n'apparaît pas libre dans `B` arrive souvent en pratique, et on peut noter `A -> B` au lieu de `forall (x:A), B`. Comme le suggèrent les notations, ces types représentent des implications et des quantifications universelles du point de vue logique. Cette construction s'appelle *produit dépendant*.

Comme les types sont des termes, il doivent également être typés. Pour cela, on introduit un ensemble de constantes appelées *sortes*. Un terme est un type de CIC s'il admet

une sorte pour type. Dans Coq, le type des formules logiques élémentaires est `Prop`. Cette sorte n'est pas suffisante : pour des raisons de cohérence logique, il est nécessaire d'ajouter une sorte pour typer `Prop` lui-même, puis une pour typer le type de `Prop`, et ainsi de suite. Le système désigne toutes ces sortes par `Type`, et s'arrange en interne pour maintenir une hiérarchie cohérente.

CIC est aussi une variante de système de types purs, ou PTS [3] (en anglais, *pure type system*), où il existe une relation d'équivalence sur les termes, appelée *conversion*, et une règle de typage (la règle de conversion) qui permet de remplacer un type par n'importe quel autre qui lui est convertible. Cette équivalence est définie à partir d'une relation de *réduction* orientée, comparable aux sémantiques opérationnelles à petits pas de langages tels que le λ-calcul. Dans CIC, la règle de conversion utilise en réalité une relation de sous-typage (non symétrique) plus large que la conversion. Une règle fondamentale de la relation de réduction est la β-réduction :

$$\texttt{(fun (x:A) => t) u} \to_\beta \texttt{t}[\texttt{x} := \texttt{u}]$$

Il existe également d'autres règles qui seront introduites au fur et à mesure. Les règles de réduction sont essentiellement les mêmes que pour $\mathrm{CIC_B}$, et seront détaillées plus exhaustivement dans le chapitre 3. Coq fournit une commande `Compute`, qui prend en argument un terme, et le réduit autant que possible (le résultat est alors dit sous *forme normale*) :

```
Coq < Compute 18 + 24.
     = 42
     : nat
```

La correspondance de Curry-Howard [27] est profondément intégrée à Coq : les formules sont des types, et les preuves des programmes. Les règles de typage peuvent être vues comme des règles de dérivation de déduction naturelle. Cependant, il est souvent utile de distinguer les aspects logiques des aspects informatifs. En effet, pour certains types, on ne s'intéresse qu'à la propriété d'être habité : la valeur précise des habitants n'est pas pertinente. Par exemple, une fonction peut prendre en argument un nombre premier, et reposer sur sa primalité pour terminer, sans se soucier de la manière dont elle a été prouvée. Pire, la preuve de primalité peut elle-même être difficile et longue à calculer, pouvant impacter significativement les performances avec une compilation naïve. Dans ce cas, on dira que la primalité est *logique*. En revanche, cette même fonction va généralement utiliser la valeur numérique de l'entier : on dira alors que l'entier est *informatif*. Bien qu'il soit possible d'utiliser un objet informatif pour construire une propriété logique (comme « n est premier »), le contraire est soigneusement interdit par le typage. La sorte des types de données informatifs élémentaires est `Set`. Tout comme pour `Prop`, son propre type est désigné par `Type`, et le système s'arrange en interne pour distinguer les `Type` qui ont besoin de l'être. `Set` peut être vu lui-même comme une instance de `Type` plus contrainte.

Le langage décrit jusqu'à présent forme un système logique déjà très puissant, le *calcul des constructions* (CC) [13], et formait le cœur des premières versions de Coq.

2.1.2 Types inductifs

CC permet de définir les types de données usuels (entiers, paires, listes, etc.) via un encodage fonctionnel imprédicatif. Cependant, cet encodage souffre de quelques limitations aussi bien logiques que calculatoires, qui ont été résolues par l'introduction des *types inductifs* [38]. Les types inductifs fournissent un moyen générique d'étendre le langage sans compromettre la cohérence logique. Chaque déclaration inductive permet de créer de nouveaux types, ainsi que des *constructeurs* permettant de créer des valeurs dans ces nouveaux types.

Pour cela, le système maintient un environnement global de types inductifs. L'environnement est initialement vide, et est peuplé via des appels au noyau, qui vérifie au passage la bonne formation des déclarations. La commande vernaculaire `Inductive` permet de déclarer un ensemble de types mutuellement inductifs, ainsi que leur constructeurs.

Ces types n'ont pas forcément besoin de définir des structures de données récursives : parmi les types inductifs non récursifs, on trouve par exemple certains connecteurs logiques tels que définis dans la bibliothèque standard. Ainsi, `True` est défini par un type inductif à un constructeur :

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

Ici, `I` est la preuve de `True`, et est comparable à la règle d'introduction de $\top$ en déduction naturelle. On peut même montrer facilement à l'intérieur du système que toute preuve de `True` est égale (pour l'égalité de Leibniz) à `I`. De plus, on déclare explicitement que l'on souhaite considérer `True` comme une propriété logique : `True` est de type `Prop`. La disjonction `or` est définie comme suit :

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

Cette déclaration fait apparaître les *paramètres* `A` et `B`, qui peuvent être utilisés dans tout le reste de la déclaration (types de l'inductif et des constructeurs). Ce type possède deux constructeurs, et peut être utilisé via le *filtrage* : si `x` est une preuve de `or A B`, et que `M` et `N` sont des preuves de `C`, alors :

```
match x with
  | or_introl a => M
  | or_intror b => N
end
```

est une preuve de `C`. La variable `a` (resp. `b`) représente une preuve de `A` (resp. `B`) que peut utiliser `M` (resp. `N`). Au filtrage est associée une règle de réduction similaire à la β-réduction : si `x` est de la forme `or_introl u` (resp. `or_intror v`), alors le terme ci-dessus se réduit vers `M`[`a` := `u`] (resp. `N`[`b` := `v`]). En réalité, le constructeur `or_introl` appliqué à l'argument `u` se note en Coq `or_introl A B u`, mais `A` et `B` sont des annotations de type plutôt que de véritables arguments de `or_introl`, au même titre que le type apparaissant dans l'abstraction, et non pas comme des arguments du constructeur `or_introl`, malgré la syntaxe trompeuse. D'ailleurs, `A` et `B` n'apparaissent pas dans les motifs de filtrage et un mécanisme de Coq permet de les rendre implicites ailleurs (voir la section 2.2).

Le filtrage permet de faire une analyse de cas sur les inductifs ayant plusieurs constructeurs. Il est également utile pour les inductifs à un seul constructeur, comme la conjonction :

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

La syntaxe plutôt intuitive de la commande `Inductive` ne doit pas faire perdre de vue que les types des constructeurs ne sont pas arbitraires : ils sont toujours de la forme :

$$\texttt{forall}\ (x_1 : T_1), \ldots, \texttt{forall}\ (x_n : T_n), I\ A_1 \cdots A_m$$

où I est le nom de l'inductif, $A_1, \ldots, A_m$ ceux de ses paramètres, et $T_1, \ldots, T_n$ sont les types des arguments du constructeur. Le constructeur `conj` prend donc deux arguments, et le filtrage d'un terme `x` de type `and A B` permet de les récupérer :

```
match x with
  | conj a b => M
end
```

Ici, `M` peut accéder aux sous-preuves de `A` et `B` sous les noms `a` et `b`. Une déclaration similaire, mais avec des dépendances entre les paramètres et les arguments de constructeurs, est la quantification existentielle :

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  ex_intro : forall (x : A), P x -> ex A P.
```

La notation `exists x:A, Q` désigne le type `ex A (fun x:A => Q)`. Un habitant de ce type est un couple d'un terme de type `A`, et d'une preuve qu'il vérifie bien le prédicat.

Comme dans la plupart des langages de programmation, Coq dispose d'*enregistrements*. Du point de vue du noyau, il ne s'agit que de types inductifs à un constructeur. Le système fournit cependant un support particulier pour ces types, et ils peuvent être déclarés avec la commande vernaculaire `Record` :

```
Record pair (A B : Type) : Type := mk_pair {
  fst : A;
  snd : B
}.
```

Du point de vue du noyau, cette commande déclare un type inductif `pair` à un constructeur `mk_pair` prenant deux arguments (un pour chacun des champs de l'enregistrement). Elle définit de plus deux constantes `fst` et `snd` effectuant les projections, et permet de manipuler les termes de type `pair` avec une syntaxe spéciale.

Enfin, un type inductif peut n'avoir aucun constructeur, et c'est ainsi qu'est défini `False` :

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

Le filtrage correspondant n'a aucune branche et peut avoir n'importe quel type. Du point de vue logique, le filtrage de `False` correspond à la règle *ex falso quodlibet*.

2.1.3 Récursion

La principale puissance des types inductifs est la possibilité de définir des structures de données récursives, et les fonctions (récursives) qui vont avec. Le type le plus élémentaire est `nat`, les entiers naturels en représentation unaire :

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

Il existe des contraintes sur les déclarations de types inductifs, en particulier la *condition de positivité* : toutes les occurrences récursives d'un type I doivent apparaître en position strictement positive dans les types des arguments des constructeurs. Cette condition est nécessaire pour garantir la terminaison du système en l'absence de points fixes. Par exemple, elle interdit l'encodage classique du λ-calcul (pur) en syntaxe abstraite d'ordre supérieur [42] :

```
Inductive term : Set :=
  | lambda : (term -> term) -> term.
```

qui, combiné au filtrage, permettrait d'écrire des termes divergents. Ici, la première occurrence de `term` est en position négative dans le type de l'argument de `lambda`.

La déclaration d'un type inductif génère automatiquement des définitions qui peuvent servir à raisonner (ou à programmer) en utilisant le principe d'induction. Par exemple, `nat` donne lieu à la définition de `nat_ind` qui a le type suivant :

```
forall P : nat -> Prop,
  P 0 -> (forall n : nat, P n -> P (S n)) ->
  forall n : nat, P n
```

On retrouve là l'axiome de récurrence de Peano.

Il est aussi possible de définir des *points fixes*, c'est-à-dire des fonctions récursives. Par exemple, l'addition peut être écrite comme suit :

```
fix plus (n m : nat) : nat :=
  match n with
    | O => m
    | S p => S (plus p m)
  end
```

Les points fixes sont aussi soumis à une contrainte particulière de bonne formation, la *condition de garde*. Cette dernière impose la présence d'un argument de type inductif qui décroît structurellement à chaque appel récursif, afin de garantir la terminaison. À ce jour, la condition de garde idéale est toujours un sujet actif de recherche, comme l'atteste par exemple [25]. Une règle de réduction permet de réduire les points fixes, et forme avec la réduction du filtrage la *ι-réduction*. Comme CIC autorise les réductions sous des contextes arbitraires en général, la ι-réduction impose que le symbole de tête de l'argument décroissant soit un constructeur avant de pouvoir déplier une définition de point fixe. Les principes d'induction évoqués précédemment ne sont en réalité que des squelettes de points fixes génériques où les appels récursifs sont toujours effectués sur les sous-termes immédiats.

Il est possible de définir en Coq toutes les fonctions récursives dont la terminaison repose sur un argument de décroissance bien fondée grâce au prédicat d'*accessibilité* :

```
Inductive Acc (A:Type) (R : A -> A -> Prop) (x:A) : Prop :=
    Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

Un terme `x` est accessible (`Acc A R x`) si tous les chemins à partir de `x` en utilisant la relation `R` inversée sont finis : la preuve d'accessibilité représente l'arbre de tous les chemins possibles. Le principe d'induction associé autorise alors les appels récursifs sur tous les successeurs accessibles. Dans le cas où `R` est bien fondée, tous les termes du type `A` sont accessibles. C'est par exemple le cas avec l'inégalité stricte sur les entiers (`lt`).

Coq propose aussi des types *co-inductifs* [21], permettant de modéliser des structures de données infinies sur lesquelles les calculs se font paresseusement. Nous ne les avons pas du tout considérés dans cette étude.

2.1.4 Indices et types dépendants

Les types inductifs permettent également de définir des objets dont une certaine propriété apparaît dans le type :

```
Inductive vect (A : Type) : nat -> Type :=
  | vnil : vect A O
  | vcons : forall (n : nat), A -> vect A n -> vect A (S n).
```

`vect A n` est le type des listes de taille `n`. `vect` lui-même est de type `Type -> nat -> Type`, mais le premier `Type` et `nat` ont deux statuts différents : comme pour `or`, le premier est un *paramètre*, constant dans les conclusions des constructeurs, alors que `nat` peut être instancié par une expression arbitraire dans les types des constructeurs, appelée *indice* ou *argument*. `vect` est un exemple de type dépendant : il dépend d'un terme informatif (de type `nat`). C'est également un type *polymorphe* (notion déjà présente dans le système F et ML) : il dépend d'un type.

La différence entre paramètre et indice est fondamentale dans la théorie, même si la syntaxe ne le laisse pas toujours transparaître. On peut voir un paramètre comme une annotation sur le type non spécifique à une valeur particulière (le type des éléments d'une liste), et un indice comme une annotation sur la valeur elle-même (la taille d'une instance particulière). La différence apparaît dans la forme complète du filtrage :

```
match x as x0 in vect _ n0 return
  match n0 with
    | O => vect nat O
    | S p => nat
  end
with
  | vnil => vnil nat
  | vcons n1 y ys => y
end
```

Dans cet exemple, on suppose que `x` est de type `vect nat n`. Les annotations (mots-clés `as`, `in` et `return`) ajoutées au premier filtrage dénotent en fait une fonction :

```
fun (n0 : nat) (x0 : vect nat n0) => match n0 ... end
```

appelée *prédicat de retour*, qui donne le type de retour du filtrage en fonction de ce qui est filtré et de ses indices (mais pas des paramètres). En présence d'indices, la forme générale du type d'un constructeur est en fait :

$$\texttt{forall}\ (x_1 : T_1), \ldots, \texttt{forall}\ (x_n : T_n), I\ A_1 \cdots A_m\ U_1 \cdots U_k$$

où $U_1, \ldots, U_k$ sont les instanciations des indices pour le constructeur considéré.

Pouvoir réaliser le produit dépendant avec `Type` confère à CIC une forme de polymorphisme comparable à celui du système F ou de ML, mais toujours explicite. Rappelons qu'en réalité, chaque occurrence de `Type` désigne une variable d'univers fixée une fois pour toutes et, *a priori*, il n'est pas possible d'écrire une fonction qui accepterait n'importe quel autre `Type` en argument. Toutefois, lorsqu'un type inductif possède un paramètre de type `Type`, une forme supplémentaire de polymorphisme lui est conférée : le paramètre peut être instancié par *n'importe quel univers*. C'est ce qu'on appelle *polymorphisme de sorte* (ou *polymorphisme d'univers*).

Le choix du statut de paramètre ou d'indice pour un argument apparent d'un inductif peut être crucial. L'interaction entre paramètres et indices dans le typage permet de définir l'égalité de Leibniz avec un type inductif :

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  eq_refl : eq A x x.
```

Ici, `eq` semble posséder deux arguments de type `A`, mais l'un est un paramètre (fixe dans le prédicat de retour) et l'autre un indice (argument du prédicat de retour). Si l'on dispose d'une preuve `H` d'égalité entre deux types `X` et `Y`, le filtrage sur cette égalité, combiné au prédicat de retour, permet le transtypage du type `X` vers le type `Y` :

```
match H in (eq _ _ Z) return (X -> Z) with
| eq_refl => fun x : X => x
end
     : X -> Y
```

L'unique constructeur, sans arguments, impose une contrainte de convertibilité entre le paramètre et l'indice. Toutefois, lorsque `x` et `y` ne sont pas clos (par exemple, les arguments d'une fonction), il peut exister des preuves de `eq A x y` où `x` et `y` ne sont pas directement convertibles. Un des principes d'induction automatiquement générés généralise ce transtypage :

```
eq_rect =
fun (A : Type) (x : A) (P : A -> Type)
    (f : P x ) (y : A) (e : eq A x y ) =>
match e in (eq _ _ y0) return (P y0) with
| eq_refl => f
end
     : forall (A : Type) (x : A) (P : A -> Type),
       P x -> forall y : A, eq A x y -> P y
```

Il s'agit du principe de Leibniz pour l'égalité.

2.1.5 Séparation logique-informatif

Dans les exemples précédents, tous les connecteurs logiques ont été déclarés dans la sorte `Prop` ; le type de données `nat` a été quant à lui déclaré dans la sorte `Set`. Déclarer un inductif dans `Prop` a un effet particulier : ses habitants vont être effacés par l'extraction. Les termes dont le type est dans `Prop` sont dits *logiques* et les autres sont dits *informatifs*. Comme l'extraction ne devrait effacer que du code mort, des restrictions existent au niveau du typage. Ainsi, il n'est en général pas possible de construire un terme informatif en filtrant un terme logique :

```
Coq < Check (fun (H : or A B) =>
Coq <   match H with
Coq <     | or_introl _ => 0
Coq <     | or_intror _ => S 0
Coq <   end).
Error:
Incorrect elimination of "H" in the inductive type "or":
the return type has sort "Set" while it should be "Prop".
Elimination of an inductive object of sort Prop
is not allowed on a predicate in sort Set
because proofs can be eliminated only to build proofs.
```

Dans cet exemple, on essaie de construire un `nat` dépendant de la manière dont `or A B` a été prouvé, ce qui empêcherait d'effacer le corps de cette preuve lors de l'extraction. Cependant, il est parfois légitime de vouloir savoir quel « côté » d'une disjonction est vrai et de s'en servir à des fins informatives, mais pour cela, il faut utiliser un autre type inductif :

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

Cette déclaration est essentiellement la même que pour `or`, mais la sorte du type est cette fois-ci `Set`. Les paramètres `A` et `B`, eux, sont toujours logiques. Il existe deux autres variantes de la disjonction, `sumor` (avec un paramètre dans `Prop`, l'autre dans `Type`) et `sum` (avec les deux paramètres dans `Type`) :

```
Inductive sumor (A : Type) (B : Prop) : Type :=
  | inleft : A -> sumor A B
  | inright : B -> sumor A B.

Inductive sum (A B : Type) : Type :=
  | inl : A -> sum A B
  | inr : B -> sum A B.
```

Toutes ces variantes se comportent identiquement au niveau de la réduction de CIC, mais possèdent différents degrés d'informativité et seront extraits différemment. La conjonction et la quantification existentielle se déclinent également en plusieurs variantes, selon les sortes apparaissant dans leur déclaration. Les principes d'induction générés automatiquement lors de la déclaration d'un inductif se déclinent également en plusieurs versions, selon la sorte de retour du prédicat.

Il existe des cas particuliers d'inductifs logiques pouvant servir à construire un terme informatif : ceux sans constructeurs (comme `False`) et ceux à un seul constructeur dont

tous les arguments sont logiques (comme `True`, `Acc` et `eq`). Ces cas particuliers sont autorisés car les inductifs ayant cette forme ne permettent pas véritablement d'encoder d'informations exploitables depuis la partie informative d'un terme.

L'extraction n'est pas la seule raison d'être de `Prop` : par défaut, cette sorte est la seule imprédicative, et les contraintes de filtrage assurent également la cohérence logique. Nous ne nous intéressons pas à ces aspects dans le cadre de l'extraction.

2.1.6 Constantes et définitions locales

En plus des types inductifs, l'environnement global contient aussi des *constantes*, qui sont des termes et des variables libres nommés et typés. Par exemple, on peut donner un nom à un terme (on parle alors de *définition*), et c'est ainsi qu'est définie la négation dans Coq :

```
Definition not := fun (A : Prop) => A -> False.
```

Il est ensuite possible d'écrire juste le nom `not` à la place du corps. Ceci est une généralisation des définitions locales déjà présentes dans CIC, particulièrement utiles pour factoriser de grosses expressions ou tout simplement organiser le code :

```
let x := A in B
```

Les règles δ (resp. ζ) permettent de déplier les définitions globales (resp. locales) lors de la réduction. Il est important de noter que, dans l'exemple ci-dessus, le corps de `x` peut être utilisé lors du typage de `B`, qui inclut la réduction ; l'expression n'est donc pas équivalente à :

```
(fun x => B) A
```

du point de vue du typage, même si les deux se réduisent vers un terme commun. En effet, dans la deuxième expression, la variable `x` doit rester abstraite lors du typage de la fonction. Par exemple, le terme :

```
let T := Type in (fun (A : T) => A -> A)
```

est bien typé alors que le terme suivant ne l'est pas :

```
(fun T => fun (A : T) => A -> A) Type
```

Une constante peut ne pas avoir de corps, mais doit toujours être bien typée (on parle alors d'*axiome*) :

```
Axiom faux : forall (A : Prop), A.
```

Cet exemple montre que la déclaration d'un axiome est à faire avec précaution : il peut introduire ce qui semble être une incohérence logique. Cependant, certains axiomes sont utiles en pratique (par exemple, le tiers exclus), et leur introduction ne pose pas de problème de cohérence. Il est aussi possible de déclarer plusieurs axiomes du même type en une fois :

```
Axioms A B C : Prop.
```

Il est évident que `Prop` est habité ; cet usage sert surtout à des fins de démonstration, pour illustrer un raisonnement qui est indépendant du choix de la valeur de `A`, `B` et `C`. Dans ce cas, on parle parfois de *paramètres*. Les commandes `Parameter` et `Parameters` sont des synonymes de `Axiom` et `Axioms`.

On peut voir les axiomes comme une généralisation des fonctions : toute la suite du développement devient une fonction acceptant une hypothétique preuve de l'axiome. Il existe d'ailleurs dans Coq un mécanisme de *section* qui concrétise cela :

```
Section introduction.
  (* ... *)
End introduction.
```

À l'intérieur de la section, les commandes `Variable` et `Variables` permettent de déclarer des axiomes locaux à la section, qui sont alors appelés *variables de section*. La commande `End` de fermeture de section va alors automatiquement transformer toutes les définitions de la section en fonctions prenant en argument les variables et les supprimer de l'environnement ; c'est la *décharge*.

Une définition peut être déclarée *opaque*, c'est-à-dire que son corps est totalement ignoré par le système, et elle joue alors le même rôle qu'un axiome (mais ne peut pas introduire d'incohérence). C'est souvent le cas des théorèmes : une fois un résultat prouvé, on en oublie la preuve. Une définition est dite *transparente* lorsqu'elle n'est pas opaque.

Tout comme `Inductive`, les commandes vernaculaires `Definition` et `Axiom` (et leurs variantes) font appel au noyau pour enrichir le langage disponible pour écrire des termes. Il est possible d'afficher le corps d'une définition grâce à la commande `Print` :

```
Coq < Print mult.
mult =
fix mult (n m : nat) : nat := match n with
                              | 0 => 0
                              | S p => m + mult p m
                              end
     : nat -> nat -> nat

Argument scopes are [nat_scope nat_scope]
```

Cette commande se généralise à toutes les entités globales : inductifs, constructeurs, axiomes...

Coq fournit également un moyen de structuration de l'environnement global avec des *modules* [11, 43]. Notre étude ignore les modules et suppose un environnement global aplati, sans sections ouvertes. D'un point de vue logique, un fichier `.vo` est une liste de déclarations d'inductifs et de constantes.

2.2 Pré-typage

Fidèlement au principe de de Bruijn, les termes CIC vus par le noyau sont annotés autant que possible, afin de faciliter la mise en œuvre de la vérification de types. Ces termes peuvent donc être très verbeux, comme le montre l'exemple de forme complète du filtrage donné précédemment. Néanmoins, le système dispose également de divers mécanismes externes au noyau permettant de compléter des termes incomplets, et en particulier l'inférence de types.

En pratique, le type de l'argument d'une fonction peut souvent être inféré à partir de l'utilisation qui en est faite et il n'est alors pas nécessaire de le donner explicitement :

```
Coq < Check (fun n => plus n O).
fun n : nat => plus n O
     : nat -> nat
```

De façon similaire, dans les filtrages, le prédicat de retour est en réalité toujours présent (même s'il n'est pas affiché), mais est inféré automatiquement dans la plupart des cas n'impliquant pas de types dépendants. Coq accepte aussi des syntaxes alternatives encore plus légères pour certains cas particuliers de types inductifs à un seul ou deux constructeurs. Ainsi, pour des propositions `A`, `B` et `C`, on peut écrire :

```
Coq < Check (fun (H : and A B) => let (a, b) := H in a).
fun (H : and A B) => let (a, _) := H in a
     : and A B -> A

Coq < Check (fun (H : or A B) (p1 p2 : C) =>
Coq <   if H then p1 else p2).
fun (H : or A B) (p1 p2 : C) =>
  if H then p1 else p2
     : or A B -> C -> C -> C
```

L'inférence de type est utilisable bien au-delà de quelques constructions syntaxiques intégrées au langage : il est possible d'y faire appel explicitement en utilisant le tiret bas (`_`) là où un terme est attendu. Par exemple, en utilisant le type inductif `vect` déclaré précédemment, on peut écrire :

```
Coq < Check (vcons _ _ 0 (vnil _)).
vcons nat 0 0 (vnil nat)
     : vect nat 1
```

En réalité, pour ce genre de situations, il existe un mécanisme d'arguments *implicites* permettant de se passer complètement des traces de sous-termes souvent inférables. Ainsi, en déclarant les deux premiers arguments de `vcons`, et l'unique argument de `vnil` comme implicites, il serait possible d'écrire directement `vcons 0 nil`. Il est toujours possible de rendre tous les arguments d'une entité globale `ident` explicites grâce à la notation `@ident`. Rappelons-le, le noyau voit toujours tous les arguments complètement renseignés.

Parmi les fonctionnalités que nous ne détaillerons pas ici, on peut citer les structures canoniques et les classes de types [45], qui permettent d'agir dans une certaine mesure sur l'inférence. Il existe également un mécanisme de coercions implicites qui peut donner l'illusion qu'un terme a plusieurs types.

2.3 Notations

Jusqu'à présent, nous avons noté tous les termes dans un style purement applicatif, mais Coq fournit un système de notations permettant d'étendre dynamiquement la grammaire des termes. Cela peut se faire notamment avec la commande vernaculaire `Notation` :

```
Notation "x + y" := (plus x y)
  (at level 50, left associativity).
```

De nombreuses notations sont définies dans la bibliothèque standard, en particulier pour les connecteurs logiques présentés précédemment :

Notation	Syntaxe primitive
`A \/ B`	`or A B`
`A /\ B`	`and A B`
`{A} + {B}`	`sumbool A B`
`exists x:A, B`	`ex A (fun x:A => B)`
`x = y`	`eq _ x y`

Dans le cas de `=`, on exploite le pré-typage pour inférer le premier paramètre de `eq`. Il existe d'autres notations qui ne rentrent pas dans le champ d'application de cette commande, comme les nombres et les chaînes de caractères. Par défaut, les notations sont aussi utilisées par Coq pour l'affichage.

Le dépliage des notations est effectué directement lors de l'analyse lexicale et syntaxique, et le noyau n'a même pas conscience de leur existence (au contraire des définitions). Le pré-typage n'est pas sollicité lors de la définition d'une notation, mais après

chaque dépliage. Une notation peut donc générer des termes différents à chaque utilisation. Néanmoins, les constantes référencées dans le corps d'une notation sont toujours liées lexicalement. Les notations, combinées au pré-typage, peuvent donner une apparence très expressive et concise (voire surprenante) à la syntaxe concrète du langage, qui le rapproche plus des mathématiques qu'un langage de programmation ordinaire. Cela peut poser des problèmes d'adéquation entre ce qui est visible et ce qui est compris par le système. L'utilisation des notations n'est jamais obligatoire pour écrire des termes, et, au besoin, il est possible d'expliciter les arguments implicites, les coercions et de ne pas utiliser les notations à l'affichage.

2.4 Tactiques

Les constantes peuvent être définies en donnant directement des termes CIC. Coq fournit aussi un mécanisme pour construire ces termes de façon interactive. Certaines commandes vernaculaires mettent le système en « mode preuve », où de nouvelles commandes, les *tactiques*, sont acceptées. Elles permettent d'écrire des preuves dans un style à la LCF [40] :

```
Lemma impl_transitivity : forall A B C,
  (A -> B) -> (B -> C) -> (A -> C).
Proof.
  intros A B C Hab Hbc a.
  apply Hbc.
  apply Hab.
  assumption.
Qed.
```

Ici, on déclare une constante `impl_transitivity` en construisant interactivement son corps. `Lemma` passe en mode preuve, `Proof` est décorative, et `Qed` soumet le terme construit au noyau et enregistre la constante (de manière opaque) dans l'environnement. Les commandes intermédiaires sont les tactiques et forment le *script de preuve*. Il peut être exécuté pas à pas, offrant une interactivité avec l'utilisateur.

Les tactiques offrent un style impératif de construction de preuves. Elles peuvent faire appel à des outils externes, et ne fournissent aucune garantie de terminaison, ni même de déterminisme ou de reproductibilité. Elles ne sont exécutées qu'en mode interactif ou lors de la compilation d'un fichier `.v` : les termes construits sont stockés dans les fichiers `.vo`, et ne sont pas reconstruits lors d'un chargement ultérieur.

Le noyau ignore l'existence des tactiques. D'un point de vue logique, ces dernières n'ont pas besoin d'une fiabilité aussi forte que pour le noyau. Les tactiques forment un langage à part entière, appelé $\mathcal{L}_{\text{tac}}$. La commande vernaculaire `Ltac` permet à l'utilisateur de facilement définir de nouvelles tactiques adaptées à ses besoins.

2.5 Greffons

L'ensemble des commandes vernaculaires utilisables interactivement ou dans un fichier `.v` forme un langage de spécification appelé *Gallina*. Il offre des possibilités de personnalisation intrinsèques, notamment avec les notations et les tactiques.

Ce langage peut également être facilement étendu dans de nombreux aspects avec des modules externes. Ces greffons (en anglais, *plug-ins*) peuvent ajouter des commandes vernaculaires qui peuvent agir sur beaucoup d'aspects de Coq, sans pour autant le compromettre (conceptuellement) du point de vue logique.

Les greffons peuvent par exemple créer des tactiques. Ils peuvent utiliser le noyau et le pré-typage, ajouter des déclarations d'inductifs ou de constantes à l'environnement global (déclarations toujours vérifiées par le noyau), faire appel à des outils externes, ou enregistrer des informations complémentaires dans les fichiers `.vo`. Il est possible de faire en sorte qu'un fichier `.vo` dont la compilation a utilisé un greffon soit utilisable sans charger de greffon.

2.6 Vérificateur autonome

La grande souplesse apportée par les greffons peut facilement éveiller la suspicion : même si le noyau est sûr, comment s'assurer qu'il n'a pas été corrompu par un greffon ? Coq fournit pour cela un vérificateur autonome, `coqchk`, embarquant un noyau minimaliste et de quoi lire les fichiers `.vo` produits par la compilation. `coqchk` vérifie la bonne formation structurelle des fichiers, ainsi que le typage de toutes les déclarations d'inductifs et de constantes et peut localiser et afficher tous les axiomes utilisés. Il ne charge aucun greffon.

2.7 Extraction

L'extraction de programmes à partir de preuves mise en œuvre dans Coq travaille directement sur les termes CIC. Elle utilise la séparation logique-informatif vue à la section 2.1.5 afin d'éliminer les sous-termes logiques d'une définition. Elle ne joue pas elle-même de rôle logique dans le noyau de Coq. Toutefois, elle est utilisée dans le but de générer des programmes certifiés et intervient donc de manière cruciale dans la chaîne de confiance et mérite, lorsqu'elle est utilisée, la même attention que le noyau. Elle est implantée concrètement sous la forme d'un greffon introduisant des commandes vernaculaires.

2.7.1 Termes purement informatifs

La principale commande est `Extraction`. Elle prend en argument le nom d'une constante, et affiche sa version extraite dans un langage au choix parmi OCaml [32] (par défaut), Haskell [41] ou Scheme [28] :

```
Coq < Extraction plus.
(** val plus : nat -> nat -> nat **)

let rec plus n m =
  match n with
  | O -> m
  | S p -> S (plus p m)
```

Dans le cas de `plus`, il n'y a aucun sous-terme logique, donc l'extraction est essentiellement un changement de syntaxe. Dans le cas où le langage cible est statiquement typé, l'extraction de Coq s'arrange aussi pour que le programme généré soit bien typé. Ainsi, le langage cible peut fournir certaines garanties (comme l'absence d'erreur à l'exécution), indépendamment de la correction de l'extraction. La commande `Extraction` fonctionne donc également sur tous les types :

```
Coq < Extraction nat.
type nat =
| O
| S of nat
```

Toutefois, le typage des programmes générés n'est pas toujours possible. En effet, il est par exemple possible de définir dans CIC le type des fonctions prenant n arguments entiers :

```
Coq < Fixpoint nary n :=
Coq <   match n with
Coq <     | O => nat
Coq <     | S p => nat -> nary p
Coq <   end.
nary is recursively defined (decreasing on 1st argument)
```

Ce type n'a pas d'équivalent en OCaml (ou en Haskell), et un type générique lui est substitué dans l'extrait. De même, il est possible de définir dans CIC une fonction acceptant un premier entier n puis ensuite $n + 1$ autres arguments entiers, et renvoyant leur somme :

```
Coq < Fixpoint sum n a : nary n :=
Coq <   match n with
Coq <     | O => a
Coq <     | S p => fun b => sum p (a+b)
Coq <   end.
sum is recursively defined (decreasing on 1st argument)
```

La fonction peut être exécutée au sein de Coq :

```
Coq < Compute sum 2 1 2 3.
     = 6
     : nat
```

Une telle fonction n'est pas typable en OCaml ou en Haskell ; l'extraction est alors obligée de contourner le système de types du langage cible. Pour OCaml, cela est fait en utilisant l'identité non typée (`Obj.magic`) :

```
Coq < Extraction sum.
(** val sum : nat -> nat -> nary **)

let rec sum n a =
  match n with
  | O -> Obj.magic a
  | S p -> Obj.magic (fun b -> sum p (plus a b))
```

Les propriétés de correction traditionnellement assurées par le typage du langage cible ne sont alors plus garanties. L'interaction entre extraction et typage dans le langage cible a été étudiée en détail dans [33], et nous ne nous y intéresserons pas davantage ici.

2.7.2 Effacement des sous-termes logiques

Le principal intérêt de l'extraction est l'effacement des sous-termes logiques d'un terme CIC. Ces sous-termes, en général nécessaires au typage, correspondent à du code mort si on ne s'intéresse qu'aux valeurs informatives. Nous avons vu à section 2.1.5 que le noyau assurait une séparation stricte entre logique et informatif, et que le choix des sortes dans la déclaration d'un type inductif était important, et que différents choix peuvent donner lieu à différents inductifs.

Les quatre variantes de la disjonction (`or`, `sumbool`, `sumor` et `sum`) se distinguent par le choix des sortes. Les règles de réduction associées sont similaires ; elles ont le même contenu calculatoire. Mais leurs extraits sont bien différentes. La version purement logique est effacée à l'extraction et n'a donc pas d'existence dans le langage cible :

```
Coq < Extraction or.
(* or : logical inductive *)
(* with constructors : or_introl
or_intror *)
```

La version dans `Set` où les paramètres sont tous les deux dans `Prop` donne un type de données isomorphe aux booléens :

```
Coq < Extraction sumbool.
type sumbool =
| Left
| Right
```

La version dans `Type` où un des paramètres est dans `Type` et l'autre dans `Prop` donne un type de données isomorphe au type optionnel (`option` en OCaml, et `Maybe` en Haskell) :

```
Coq < Extraction sumor.
type 'a sumor =
| Inleft of 'a
| Inright
```

Enfin, la version dans `Type` où tous les paramètres sont dans `Type` donne un type de données assimilable à l'union disjointe de deux ensembles :

```
Coq < Extraction sum.
type ('a, 'b) sum =
| Inl of 'a
| Inr of 'b
```

Même si nous ne nous intéressons pas au typage dans le langage cible ici, ces cas illustrent bien que l'extraction peut être une opération non triviale lorsque termes logiques et informatifs sont mélangés. Plus concrètement, il est possible de définir une fonction de prédécesseur fortement contrainte, qui exige que son argument soit non nul, et qui en contrepartie garantit la propriété voulue entre son résultat et son argument :

```
Definition pred : forall n, n <> 0 -> { p | n = S p }.
```

La notation `{ p | n = S p }` désigne la variante (`sig`) de quantification existentielle dans `Type` où l'argument du prédicat est dans `Type`. Autrement dit, l'extraction va conserver le `p`, mais effacer la preuve de `n = S p`. La constante `pred` peut être définie par tactiques, ou directement via un terme CIC tel que :

```
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 (eq_refl 0) return {p : nat | 0 = S p} with
    end
| S n0 =>
    fun _ : S n0 <> 0 =>
    exist (fun p : nat => S n0 = S p) n0 (eq_refl (S n0))
end H
```

La fonction est alors extraite comme suit :

```
(** val pred : nat -> nat **)
let pred = function
| O -> assert false (* absurd case *)
| S n0 -> n0
```

Cet exemple illustre quatre aspects différents de l'extraction :

- l'argument logique de `pred` — l'hypothèse `n <> 0` — a été complètement effacé ; les `fun` correspondants ont été supprimés ;
- le filtrage sur un inductif vide — la preuve de `0 <> 0` (convertible à `0 = 0 -> False`) appliquée à `eq_refl 0` — a été remplacé par une erreur à l'exécution ; la cohérence logique du système assure qu'elle ne sera jamais déclenchée (avec une réduction faible) par du code issu de Coq ;
- le premier argument apparent du constructeur `exist`, qui est en fait un paramètre de l'inductif `sig` et, à ce titre, peut être considéré comme une annotation de type, est effacé. De manière générale, toutes les annotations de type sont effacées ; l'éventuel typage du programme généré est refait *a posteriori* à partir de zéro ;
- l'argument logique de `exist` (la preuve de `S n0 = S n0`) a été effacé.

Un cinquième aspect non représenté ici est l'effacement des filtrages sur un inductif logique singleton.

L'effacement du deuxième argument de `pred` ne pose pas de problème pour un programme entièrement écrit en Coq dans la mesure où toutes les utilisations de `pred` sont adaptées par l'extraction :

```
Coq < Definition piege := pred 0.
piege is defined

Coq < Extraction piege.
(** val piege : __ -> nat **)

let piege _ =
  pred O
```

Cependant, des problèmes peuvent se présenter en cas d'interfaçage avec du code écrit manuellement, comme le déclenchement d'une erreur à l'exécution dans le cas d'un appel naïf à `pred 0`.

Le traitement du filtrage sur un inductif vide montre bien l'importance de la cohérence logique du système pour l'extraction. De plus, il est important que le langage cible n'autorise pas des réductions arbitraires dans tous les contextes, comme c'est le cas dans dans CIC : on ne veut pas que le `assert false` de l'exemple ci-dessus soit exécuté prématurément ! Plus formellement, il faut que le langage cible n'autorise que les réductions de termes clos (la réduction est alors qualifiée de *faible*), et c'est le cas des langages cibles de l'extraction de Coq, ainsi que du langage cible de l'extraction étudiée au chapitre 3. En particulier, des axiomes peuvent permettre à l'extraction de générer des programmes erronés. Certes, le cas du filtrage d'une preuve de `False` est détectable et géré gracieusement, mais un programme extrait peut, dans un contexte incohérent, planter de façon plus spectaculaire sans que cela ne soit détectable statiquement. Par exemple, une preuve d'égalité permet un transtypage sans vérification :

```
Coq < Extraction eq_rect.
(** val eq_rect : 'a1 -> 'a2 -> 'a1 -> 'a2 **)

let eq_rect x f y =
  f
```

Ainsi, une preuve de `nat = nat -> nat` permettrait de réaliser une erreur de segmentation. Un autre exemple d'erreur pouvant apparaître dans un programme extrait est la non-terminaison, s'il utilise une fonction récursive dont l'argument inductif décroissant est logique (comme avec `Acc`).

2.7.3 Conclusion

L'extraction que nous allons étudier en détail dans le chapitre 3 n'est pas aussi avancée que celle de Coq : les sous-termes logiques laissent toujours une trace dans les programmes générés, et on suppose (comme dans Coq) que ces derniers sont exécutés avec des entrées qui vérifient les hypothèses qui auraient été éventuellement effacées par l'extraction. Nos résultats de correction ne s'appliquent qu'à des termes clos.

En pratique, il est possible de toujours se passer de `Prop`.[1] Mais l'utilisateur (ou plutôt, le programmeur) a tout intérêt à maximiser la partie logique de ses fonctions Coq tout en gardant informatif ce qui doit l'être, dans le but d'optimiser la concision et les performances des fonctions obtenues par extraction.

1. sauf si l'imprédicativité de `Prop` est utilisée

3 Extraction certifiée

Ce chapitre présente nos travaux de formalisation en Coq de l'extraction. Il est important pour la suite de bien faire la distinction entre l'extraction objet de l'étude, que nous qualifierons de *formelle*, et l'extraction déjà implantée dans Coq, que nous qualifierons d'*informelle*.

Nos travaux font suite à ceux de B. Barras [5] qui, à la fin des années 1990, a formalisé un langage proche de CIC, que nous appellerons $\mathrm{CIC_A}$. B. Barras s'est intéressé à la méta-théorie du système de types, et a prouvé notamment la décidabilité du typage. L'extrait informel de ce développement a produit ce qui est comparable au noyau de Coq, et a été interfacé avec du code OCaml écrit manuellement pour réaliser un nouvel assistant de preuve minimaliste et autonome, Bcoq.

$\mathrm{CIC_A}$ n'autorisait pas les cas particuliers de filtrage sur un inductif logique permettant de construire des termes informatifs (voir la section 2.1.5). Ces cas particuliers étant importants pour l'extraction, nous avons dû modifier le système de types pour en tenir compte. Nous avons aussi choisi d'expliciter ces cas particuliers au niveau de la syntaxe. Les modifications que nous avons dû apporter au langage de B. Barras donnent techniquement un nouveau langage, que nous appelons $\mathrm{CIC_B}$.

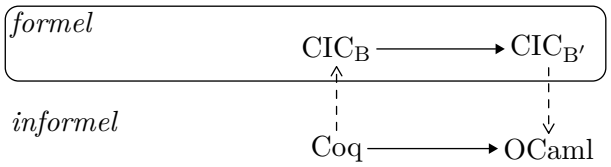
Nous donnons dans la section 3.1 une présentation unifiée de $\mathrm{CIC_B}$. Les définitions et résultats sont essentiellement tirés de [5]. Nos contributions sont, outre la présentation elle-même, l'ajout des cas particuliers de filtrage, et l'identification d'un lemme général sur $\mathrm{CIC_B}$ inutile dans le développment de B. Barras, mais nécessaire pour notre formalisation de l'extraction.

La section 3.2 traite de l'extraction proprement dite, et contient la majeure partie de nos contributions. En nous inspirant des travaux de P. Letouzey (voir la section 2.3 de [33]), nous avons défini une *relation* d'extraction qui consiste à remplacer des sous-termes logiques par une constante. Nous avons prouvé que les réductions dans le langage cible pouvaient bien être simulées dans le langage source, et réciproquement. La relation d'extraction permet de définir facilement une *fonction* d'extraction certifiée, que nous avons intégrée à Bcoq (section 3.3.1). Nous proposons également une application nouvelle (section 3.3.2) de l'extrait informel du développement.

Nous discutons dans la section 3.4 de certains aspects de l'extraction de Coq qui sortent du cadre de notre étude, mais qui méritent néanmoins une attention particulière. Enfin, nous dressons dans la section 3.5 un bilan du travail réalisé.

Notre extraction formelle vise un langage cible abstrait, $\mathrm{CIC_{B'}}$, et se veut être un tronc commun pour une hypothétique extraction formelle vers les langages supportés

par l'extraction informelle. Notre développement peut être mis en regard du sytème Coq existant :



Le monde formel représente tout ce qui est écrit en Coq, et le monde informel représente tout ce qui est écrit en OCaml. Les passerelles entre les deux mondes n'ont pas été réalisées, faute de temps.

$\mathrm{CIC_B}$ est complexe, et cette complexité est présente dans le développement Coq. Nous avons tenté d'y rester fidèle dans ce chapitre, mais certaines définitions et propositions formelles ont été enjolivées et peuvent parfois s'éloigner du développement Coq.

3.1 Le calcul des constructions inductives (CIC)

Nous allons présenter dans cette section le langage source de notre extraction. Ce langage est entièrement formalisé en Coq. Il s'agit d'une variante du calcul des constructions inductives comparable au noyau de Coq, que nous appelons $\mathrm{CIC_B}$. Cette section est une reformulation synthétique des travaux de B. Barras [5], et peut servir de complément formel à la section 2.1.

3.1.1 Syntaxe

Soit `name` un ensemble de *noms*, c'est-à-dire un type de données quelconque, infini, dénombrable et dont l'égalité est décidable.

Définition 3.1 (type `term`)**.** Les *termes* sont définis par la grammaire suivante :

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

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

Cette syntaxe inclut de manière primitive les variables (on utilise la représentation de de Bruijn [18]) ; l'abstraction et son type, le produit dépendant, utilisé pour représenter les quantifications universelles et les implications dans la logique ; les paires non dépendantes et leur type ; les sortes et les opérateurs (un opérateur pouvant être constant ou appliqué). L'abstraction et le produit dépendant sont annotés d'un nom, utilisé comme suggestion de nommage de la variable liée dans les interactions avec l'utilisateur ou pour

le confort du lecteur. L'égalité sur les termes est considérée modulo ces noms. On notera $t\{\natural n \leftarrow u\}$ le terme t dans lequel la variable n a été remplacée par u (en adaptant les variables libres de t et de u en conséquence), et $\uparrow_k^n t$ le terme t dans lequel les variables libres plus grandes que k ont été augmentées de n (k sera omis si nul).

La syntaxe des termes fait appel aux notions de sortes et d'opérateurs.

Définition 3.2 (type `sort`)**.** Les *sortes* sont des constantes servant à typer les types. Elles sont données par la grammaire suivante :

$$s : \texttt{sort} \quad ::= \quad \mathsf{Prop} \mid \mathsf{Set} \mid \mathsf{Type}_n$$

où $n \in \mathbb{N}$.

Prop représente le type des propositions logiques dont on éliminera les preuves lors de l'extraction. Tous les autres types sont de type Set ou Type_n.

Définition 3.3 (type `oper`)**.** Les *opérateurs* sont définis par la grammaire suivante :

$$\begin{aligned} c : \texttt{oper} \quad ::= \quad & \pi_1 \mid \pi_2 \mid \texttt{App} \mid \texttt{Const}(C) \\ & \mid \texttt{MutInd}(I, n) \mid \texttt{MutConstr}(C) \mid \texttt{MutCase}(\beta, \vec{x}) \mid \texttt{Fixp} \\ & \mid \Box \mid \varepsilon \mid \texttt{Cons\_mark} \mid \mathcal{M} \mid \mathcal{L} \\ & \mid \texttt{Record}(\vec{x}) \mid \texttt{Struct}(\vec{x}) \mid \texttt{Field}(x) \end{aligned}$$

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

Bien que la syntaxe autorise l'utilisation de n'importe quel opérateur non appliqué, ou appliqué à n'importe quel terme, on verra par la suite que seules certaines combinaisons sont acceptées par le typage. On utilisera les notations suivantes, qui préfigurent les combinaisons possibles :

Notation	Syntaxe primitive
$\{\vec{t}\}$	$(t_1, (t_2, \ldots (t_n, \varepsilon)))$ ou $(t_1 * (t_2 * \cdots (t_n * \mathcal{L})))$
$f\ x$	$\texttt{App}((f, x))$
$\mathsf{Ind}\{I \mid p, a\}\langle n, l\rangle$	$\texttt{MutInd}(I, n)((p, (a, l)))$
$\mathsf{Constr}\{C \mid p, a\}$	$\texttt{MutConstr}(C)((p, a))$
$\mathsf{Match}_\beta\{t \Rightarrow \vec{x} \mid P \Rightarrow \vec{b}\}$	$\texttt{MutCase}(\beta, \vec{x})((t, (P, \{\vec{b}\})))$
$\mathsf{Fix}\{\vec{t} : \vec{T}\}$	$\texttt{Fixp}((\{\vec{T}\}, \lambda m : \mathcal{M}.\lambda f : \vec{T}^{-m}.\{\vec{t}\}))$
$h :: t$	$\texttt{Cons\_mark}((h, t))$
$(\!\vert \vec{x} : \vec{T} \vert\!)$	$\texttt{Record}(\vec{x})(\{\vec{T}\})$
$(\!\vert \vec{x} : \vec{T} := \vec{t} \vert\!)$	$\texttt{Struct}(\vec{x})((\{\vec{T}\}, \{\vec{t}\}))$

Ces notations correspondent aux différentes constructions évoluées de $\mathrm{CIC_B}$:

– une liste de termes quelconques ou leurs types (selon le contexte). De manière générale, on encodera des n-uplets par des paires imbriquées (avec associativité à droite) ;

- l'application au sens usuel du λ-calcul ;
- l'inductif nommé I, dont le paramètre est instancié par p et l'indice par a. $\langle n, l\rangle$ est une annotation utilisée pour la terminaison, expliquée à la section 3.1.3 ;
- le constructeur C appliqué à l'argument a, et dont le paramètre du type est p ;
- le filtrage du terme t. $\vec{x}$ est la liste des constructeurs acceptés, P est le prédicat de retour, et $\vec{b}$ est la liste des valeurs de retour selon les différents cas. La signification de β sera vue plus tard ;
- un paquet de points fixes, c'est-à-dire une liste de fonctions mutuellement récursives, avec leurs types. Les corps peuvent faire référence à une variable de marque $\natural 1$ (m) et une variable récursive $\natural 0$ (f), communes à tous les corps, et qui n'apparaissent pas dans les types (voir la section 3.1.3) ;
- une liste non vide de marques (la liste vide était ε) ;
- le type d'un enregistrement dont les champs sont étiquetés par $\vec{x}$ et ont pour types $\vec{T}$;
- un enregistrement dont les champs sont étiquetés par $\vec{x}$, ont pour types $\vec{T}$ et pour valeurs $\vec{t}$.

De plus, on utilisera directement la syntaxe primitive pour les projections (π_1 et π_2), la référence à une constante globale ($\texttt{Const}(C)$), l'unique marque concrète ($\square$), la liste vide de marques (ε), le type des marques ($\mathcal{M}$) et le type des listes de marques ($\mathcal{L}$).

On dira qu'un terme est *localement clos* s'il ne possède aucun indice de de Bruijn libre, et qu'il est *globalement clos* s'il ne fait référence à aucune constante globale. En l'absence de précision, un terme clos le sera à la fois localement et globalement.

La réduction et le typage de $\mathrm{CIC_B}$ font aussi appel aux notions d'environnement et de signature.

Définition 3.4 (type `env`)**.** Un *environnement* est une liste de variables annotées, selon la grammaire suivante :

$$\begin{array}{rcl} \Gamma : \texttt{env} & ::= & [] \\ & | & \Gamma[x : T] \\ & | & \Gamma[x \doteq t : T] \end{array}$$

Une entrée de l'environnement déclare une variable, son type et éventuellement un corps. L'ordre est important et sert de référence pour les indices de de Bruijn : $\natural 0$ représente la dernière entrée de Γ, que l'on notera $\Gamma(0)$.

En plus de ces indices, utilisés pour les lieurs primitifs de la syntaxe, $\mathrm{CIC_B}$ fait aussi appel à des noms qui sont définis dans un environnement global, ou signature. Contrairement aux annotations de l'abstraction et du produit dépendant, ces noms font partie intégrante de la théorie.

Définition 3.5 (type `sigma`)**.** Une *signature* est une liste de *déclarations globales* définie

par la grammaire suivante :

$$\begin{array}{rcl} \Sigma : \texttt{sigma} & ::= & [] \mid \Sigma[o] \mid \Sigma[\vec{i}] \\ o : \texttt{constant\_obj} & ::= & \langle C, P, t, T\rangle \mid \langle C, P, T\rangle \\ i : \texttt{one\_ind} & ::= & \langle I, P, A, r, s, c^\star\rangle \\ c^\star & ::= & \star \mid \vec{c} \\ c : \texttt{constructor} & ::= & \langle C, T, a\rangle \end{array}$$

où $a, t, P, A, T \in \texttt{term}$, $I, C \in \texttt{name}$ et $r, s \in \texttt{sort}$.

On distingue dans cette définition deux types de déclarations globales :
- une *constante* nommée C, munie de son type T, acceptant un paramètre de type P, et ayant éventuellement un corps t. Les constantes sans corps sont aussi appelées *axiomes*. Le terme $\texttt{Const}(C)(p)$ fait référence à la constante C dont le paramètre est instancié par p ; ce dernier doit donc avoir le type P ;
- un paquet d'*inductifs*, c'est-à-dire une liste de types algébriques mutuellement récursifs ; chacun d'entre eux a un nom I, accepte un *paramètre* de type P et un *indice* (ou *argument*) de type A, est lui-même de sorte r, et ses constructeurs acceptent des arguments dont le type est de sorte s. Le terme $\mathsf{Ind}\{I \mid p, a\}\langle n, l\rangle$ fait référence au type I où le paramètre a été instancié par p et l'indice par a. Un inductif peut être *abstrait* ou *concret*. Les inductifs abstraits sont utilisés pour typer les déclarations elles-mêmes et ne peuvent pas être filtrés ; on notera $i^\star$ une déclaration d'inductif rendue abstraite. Un inductif concret contient une liste de *constructeurs*, chacun muni de son nom C, et acceptant un *argument* de type T, et instanciant l'indice de son type avec a (qui doit donc être de type A). Un constructeur C se matérialise au niveau des termes sous la forme $\mathsf{Constr}\{C \mid p, t\} : p$ (de type P) est le paramètre de l'inductif, t (de type T) est l'argument du constructeur, et le tout a le type $\mathsf{Ind}\{I \mid p, a\}\langle n, l\rangle$ (pour la signification de $\langle n, l\rangle$, voir section 3.1.3).

Les déclarations globales de $\mathrm{CIC_B}$ sont légèrement différentes de celles de Coq, où les constantes n'ont pas de paramètre, les inductifs peuvent avoir plusieurs paramètres et plusieurs indices, et les constructeurs peuvent avoir plusieurs arguments. Une constante avec paramètre peut se simuler avec une fonction, et arguments multiples peuvent se simuler avec des paires non dépendantes ou des enregistrements dépendants.

On utilisera par la suite $\Sigma(x)$ pour référencer la déclaration de x, ce dernier pouvant être une constante, un inductif ou un constructeur.

3.1.2 Réduction et sous-typage

Les propriétés de correction de l'extraction prouvées dans le cadre de cette thèse s'intéressent à la réduction : nous montrons en Coq que les réductions du langage cible peuvent être simulées dans le langage source, et que les réductions faibles de tête du langage source peuvent être simulées dans le langage cible.

Nous allons maintenant définir la réduction de $\mathrm{CIC_B}$. Le prédicat `redn_term` contient les règles de réduction de tête.

Définition 3.6 (relation `redn_term`, $\to_{\mathrm{B}}$)**.** La relation `redn_term` est l'union des règles suivantes, qui dépendent d'une signature Σ et d'un environnement Γ constants, et laissés implicites :

$$\frac{\Gamma(n) = [x \doteq t : T]}{\natural n \to_{\delta} \uparrow^{n+1} t}\ \texttt{delta}$$

$$\frac{\Sigma(C) = [\langle C, p, b, T\rangle : \texttt{constant\_obj}]}{\mathtt{Const}(C)(M) \to_{\Delta} b\,\{\natural 0 \leftarrow M\}}\ \texttt{delta\_glob}$$

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

$$\frac{y = x_i}{\mathtt{Field}(y)(\!|\, \vec{x} : \vec{T} := \vec{t} \,|\!) \to_{\pi_r} t_i}\ \texttt{projr}$$

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

$$\frac{t = \mathsf{Constr}\{C \mid p', a\} \qquad x_i = C}{\mathsf{Match}_{\beta}\{t \Rightarrow \vec{x} \mid P \Rightarrow \vec{b}\} \to_{\iota_{\mathtt{case}}} b_i\, a}\ \texttt{iota\_case}$$

$$\frac{F = \mathsf{Fix}\{\vec{f} : \vec{T}\} = \mathtt{Fixp}((\{\vec{T}\}, F_0)) \qquad t = \mathsf{Constr}\{C \mid p', a\}}{\pi_1(\pi_2^n F)\ p\ t \to_{\iota_{\mathtt{fix}}} \pi_1(\pi_2^n(F_0 \,\square\, F))\ p\ t}\ \texttt{iota\_fix}$$

Les règles `delta` à `iota_case` sont tout à fait classiques pour du λ-calcul avec types algébriques. La règle `iota_fix` donne la réduction des points fixes et sera expliquée à la section 3.1.3. On remarquera que le paramètre p' des constructeurs est ignoré par la réduction, conformément à l'intuition qu'il s'agit d'une annotation de type.

Dans $\mathrm{CIC_B}$, la réduction peut aussi se passer dans n'importe quel sous-terme, ce qui peut s'obtenir en utilisant un opérateur de passage au contexte.

Définition 3.7 (opération `ctxt`)**.** Soit $\to$ une règle de réduction. Les règles suivantes définissent la *clôture par contexte* $\to_{\mathcal{X}}$ de $\to$:

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

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

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

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

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

Traditionnellement, *la* réduction de $\mathrm{CIC_B}$ est $\to_{\mathrm{B}\mathcal{X}}$, et un terme est dit en *forme normale* s'il est irréductible pour cette relation. Toutefois, c'est sa clôture réflexive, symétrique et transitive, que nous noterons $\equiv_{\mathrm{B}}$, qui joue un rôle dans le typage. On dira que deux termes M et N sont *convertibles* si $M \equiv_{\mathrm{B}} N$. Une notion plus générale, le sous-typage, est utilisée par le typage :

Définition 3.8 (relation `cci_subtype`)**.** La relation de *sous-typage* est définie par les règles suivantes, qui dépendent d'une signature Σ constante et laissée implicite :

$$\frac{t \equiv_{\mathrm{B}} t'}{\Gamma \vdash t \leqslant t'} \qquad \frac{\Gamma \vdash T_1 \leqslant T_2 \quad \Gamma \vdash T_2 \leqslant T_3}{\Gamma \vdash T_1 \leqslant T_3}$$

$$\frac{}{\Gamma \vdash \mathsf{Prop} \leqslant s} \qquad \frac{}{\Gamma \vdash \mathsf{Set} \leqslant \mathsf{Type}_n} \qquad \frac{n \leqslant m}{\Gamma \vdash \mathsf{Type}_n \leqslant \mathsf{Type}_m}$$

$$\frac{\Gamma \vdash U_1 \leqslant T_1 \quad \Gamma[x : U_1] \vdash T_2 \leqslant U_2}{\Gamma \vdash \Pi x : T_1.T_2 \leqslant \Pi x : U_1.U_2} \qquad \frac{\Gamma \vdash T_1 \leqslant U_1 \quad \Gamma \vdash T_2 \leqslant U_2}{\Gamma \vdash T_1 * T_2 \leqslant U_1 * U_2}$$

$$\frac{\Gamma \vdash T_1 \leqslant U_1 \qquad \dots \qquad \Gamma[\_ : T_1] \cdots [\_ : T_{n-1}] \vdash T_n \leqslant U_n}{\Gamma \vdash (\!|\, \vec{x} : \vec{T} \,|\!) \leqslant (\!|\, \vec{x} : \vec{U} \,|\!)}$$

$$\frac{\langle n_1, l_1 \rangle \preccurlyeq_{\mathcal{M}} \langle n_2, l_2 \rangle}{\Gamma \vdash \mathsf{Ind}\{I \mid p, a\}\langle n_1, l_1 \rangle \leqslant \mathsf{Ind}\{I \mid p, a\}\langle n_2, l_2 \rangle}\ \texttt{ind\_marks}$$

Cette définition fait apparaître une hiérarchie croissante de sortes, un produit dépendant contravariant à gauche et covariant à droite, et un produit cartésien et des types d'enregistrement covariants. La règle `ind_marks` de sous-typage des inductifs est plus complexe, et ne sera pas détaillée (le lecteur curieux pourra se référer au développement Coq ou à [5]).

3.1.3 Typage des inductifs et des points fixes

Dans $\mathrm{CIC_B}$, la terminaison des points fixes est garantie par typage, grâce des artifices syntaxiques, les *marques*, que nous allons décrire informellement.

Le type $\mathcal{M}$ des marques n'a qu'un seul habitant clos, $\Box$. Toutefois, le typage va faire intervenir des variables de type $\mathcal{M}$, sans s'intéresser à leur valeur. Lorsqu'elle apparaît dans le type d'un terme inductif, une marque capture l'origine de ce terme. Cet usage est à rapprocher des types fantômes. Les listes de marques $\mathcal{L}$, ainsi que leurs constructeurs, sont également intégrés au langage. $\mathcal{L}$ et ε sont aussi utilisés pour encoder des n-uplets (voir tableau de notations page 37) ; cette utilisation n'a rien à voir avec le typage des inductifs.

Les termes inductifs font intervenir dans leur type une liste de marques, chacune munie d'un signe (positif ou négatif). Ces marques sont typiquement des variables indiquant l'origine du terme. Chaque paquet de points fixes mutuels introduit une variable de marque. Si cette variable apparaît positivement dans une liste de marques, alors le terme associé est issu de l'argument du point fixe et doit être filtré avant d'être utilisé dans un appel récursif ; si elle apparaît négativement, alors il s'agit d'un sous-terme directement utilisable par un appel récursif. Il faut comprendre une annotation $\langle n, l\rangle$ comme la liste l où les n premières marques sont positives, et les autres négatives. Comme l est typiquement constituée de variables, il est erroné de penser qu'elle est isomorphe à un entier.

Avec ce système, les corps des points fixes sont typés dans un environnement où les types des points fixes en cours de construction ont été modifiés de sorte que les appels récursifs ne puissent se faire que sur des sous-termes stricts. Rappelons le développement de la notation $\mathsf{Fix}\{\vec{t} : \vec{T}\}$:

$$\mathtt{Fixp}((\{\vec{T}\}, \lambda m : \mathcal{M}.\lambda f : \vec{T}^{-m}.\{\vec{t}\}))$$

Ici, $\vec{T}$ représente le vecteur des types des points fixes mutuels, m correspond à la variable de marque mentionnée ci-dessus, et $\vec{T}^{-m}$ est le vecteur des types modifiés, utilisés pour typer les corps des points fixes ; il s'agit de $\vec{T}$ où une marque $-m$ a été ajoutée au type de l'argument inductif décroissant de chacun des points fixes. Rappelons maintenant la règle de réduction `iota_fix` :

$$\frac{F = \mathsf{Fix}\{\vec{f} : \vec{T}\} = \mathtt{Fixp}((\{\vec{T}\}, F_0)) \qquad t = \mathsf{Constr}\{C \mid p', a\}}{\pi_1(\pi_2^n F)\ p\ t \to_{\iota_{\mathtt{fix}}} \pi_1(\pi_2^n(F_0\ \Box\ F))\ p\ t}\ \mathtt{iota\_fix}$$

Le côté gauche $\pi_1(\pi_2^n F)\ p\ t$ représente un appel au n-ième point fixe du paquet F, appliqué à p et t (t est l'argument inductif, et p ses dépendances). Le côté droit instancie les variables m et f introduites par la notation : m — dont la valeur est sans importance pour la sémantique opérationnelle — est instanciée avec $\Box$, et f est instanciée par F lui-même. Le terme ainsi obtenu est un vecteur de points fixes « dépliés une fois », sur

lequel on effectue la n-ième projection qu'on applique ensuite à p et t. Cette règle est applicable uniquement lorsque le symbole de tête de t est un constructeur.

Ce système est différent de celui utilisé par Coq, où la propriété de décroissance est assurée par une condition de garde syntaxique. Par exemple, considérons l'identité définie récursivement sur les entiers unaires, écrite traditionnellement en Coq comme suit :

```
fix id n := match n with
              | O => O
              | S p => S (id p)
            end
```

Ce terme a le type `nat -> nat`. En Coq, le corps de `id` est typé dans un environnement où `id` a le type `nat -> nat`, et `n` a le type `nat` ; le système vérifie de plus que chaque appel à `id` est effectué sur des sous-termes de `n`. Dans $\mathrm{CIC_B}$, le terme ci-dessus s'écrirait plutôt $\mathsf{Fix}\{t : T\}$, où :

$$\begin{aligned}
\mathsf{Fix}\{t : T\} &= \mathtt{Fixp}((\{T\}, \lambda m : \mathcal{M}.\lambda f : T^{-m}.\{t\})) \\
T &= \Pi_.\Pi n : \mathtt{nat}.\mathtt{nat} \\
t &= \lambda_.\lambda n : \mathtt{nat}^{+m}.t' \\
t' &= \mathsf{Match_r}\{n \Rightarrow \mathtt{O}, \mathtt{S} \mid P \Rightarrow u_{\mathtt{O}}, u_{\mathtt{S}}\} \\
P &= \lambda_.\lambda_.\mathtt{nat} \\
u_{\mathtt{O}} &= \lambda_.\mathsf{Constr}\{\mathtt{O} \mid \varepsilon, \varepsilon\} \\
u_{\mathtt{S}} &= \lambda a : \mathtt{nat}^{-m}.\mathsf{Constr}\{\mathtt{O} \mid \varepsilon, f\ a\}
\end{aligned}$$

Nous avons noté :

- `nat`, le terme $\mathsf{Ind}\{\mathtt{nat} \mid \varepsilon, \varepsilon\}\langle 0, \varepsilon\rangle$;
- $\mathtt{nat}^{+m}$, le terme $\mathsf{Ind}\{\mathtt{nat} \mid \varepsilon, \varepsilon\}\langle 1, m :: \varepsilon\rangle$;
- $\mathtt{nat}^{-m}$, le terme $\mathsf{Ind}\{\mathtt{nat} \mid \varepsilon, \varepsilon\}\langle 0, m :: \varepsilon\rangle$.

Quelques explications s'imposent :

- les $\Pi\_$ et $\lambda\_$ représentent des liaisons de variables inutilisées (les types sont omis). Dans le cas de T et de b, il s'agit des dépendances du type de l'argument du point fixe ; dans le cas de P, il s'agit de l'argument du type de l'objet filtré, et l'objet filtré lui-même ; dans le cas de $u_{\mathtt{O}}$, de l'argument du constructeur `O` ;
- les noms m et f font référence aux variables introduites par la notation $\mathsf{Fix}\{t : T\}$;
- dans $\mathsf{Ind}\{\mathtt{nat} \mid \varepsilon, \varepsilon\}\langle 0, \varepsilon\rangle$, les deux premières occurrences de ε représentent un 0-uplet, qui encodent le fait que `nat` n'a ni paramètre, ni argument (et cette utilisation n'a donc rien à voir avec les marques), et la dernière représente une liste de marques vide ;
- $\mathtt{nat}^{+m}$ est annoté de la liste à une seule marque, m, en position positive ; dans $\mathtt{nat}^{-m}$, la marque est en position négative ;
- t a le type $T^{+m} = \Pi\_.\Pi n : \mathtt{nat}^{+m}.\mathtt{nat}$, et est typé dans un environnement où f a le type $T^{-m} = \Pi\_.\Pi n : \mathtt{nat}^{-m}.\mathtt{nat}$. Le changement de signe dans le type de

l'argument impose le passage à travers un filtrage pour déconstruire n. Le type du point fixe « vu de l'extérieur », T, ne fait pas référence à m. Les transformations $T \mapsto T^{+m}$ et $T \mapsto T^{-m}$ décrites informellement ici seront utilisées sans donner plus de détails dans les règles de typage ;
- le point fixe se réduit en instanciant m par $\Box$ et f par $\mathsf{Fix}\{t : T\}$; il faut de plus que n soit instancié par un terme ayant un constructeur (`O` ou `S`) en tête ;
- t' fait apparaître des contraintes sur le filtrage :
  - dans l'expression $\mathsf{Match}_\beta\{n \Rightarrow \vec{x} \mid P \Rightarrow \vec{b}\}$, la liste $\vec{x}$ correspond à la liste complète des constructeurs du type I de n. On notera cette condition $I \dashv \vec{x}$ (par exemple, $\texttt{nat} \dashv \texttt{O}, \texttt{S}$),
  - si on note l la liste de marques annotant le terme filtré (dans l'exemple, $l = m :: \varepsilon$), alors pour chaque constructeur C tel que $\Sigma(C) = \langle C, U, a' \rangle$, la branche correspondante est une fonction attendant un argument de type U^-, où les occurrences de $\mathsf{Ind}\{\texttt{nat} \mid p, a'\}\langle 0, \varepsilon \rangle$ (et, de manière générale, des inductifs du même paquet) ont été remplacées par $\mathsf{Ind}\{\texttt{nat} \mid p, a'\}\langle 0, l \rangle$, et renvoyant quelque chose en accord avec P. Le type de cette branche est donc $B_C = \Pi a : U^-.P\ a'\ \mathsf{Constr}\{C \mid p, a\}$. On notera $P\langle l \rangle \dashv \vec{x} : \vec{B}$ cette condition, dans le cas où la liste des constructeurs est $\vec{x}$, et la liste des types des branches est $\vec{B}$.

Cet exemple peut laisser penser que la liste de marques d'une annotation est toujours réduite à un élément, mais elle peut être plus grande dans le cas de points fixes imbriqués. Il ne contient qu'un seul inductif, mais les explications ci-dessus se généralisent aux paquets de plusieurs points fixes.

Pour utiliser l'argument inductif d'un point fixe avec des fonctions tierces, il est nécessaire de mettre en relation les types marqués avec les types non marqués. Cette mise en relation est effectuée par le sous-typage (définition 3.8), via la règle règle `ind_marks`, qui utilise une relation notée $\preccurlyeq_{\mathcal{M}}$. Cette relation permet notamment de promouvoir $\texttt{nat}^{+m}$ et $\texttt{nat}^{-m}$ en `nat`, mais également $\texttt{nat}^{-m}$ en $\texttt{nat}^{+m}$, ce qui correspond bien à l'intuition des sous-termes.

La manière dont la terminaison des points fixes est assurée (marques ou condition de garde syntaxique) est marginale dans le cadre de l'extraction, et nous n'entrerons pas plus dans les détails. Le principe du typage avec marques est détaillé dans [5].

3.1.4 Typage des termes et des environnements

Le typage est exprimé à l'aide de deux prédicats mutuellement inductifs, `wf` (typage des environnements) et `typ` (typage des termes), qui sont caractéristiques des PTS et que nous allons décrire en premier. Ces prédicats sont constitués de règles faisant appel à des relations auxiliaires que nous allons décrire au fur et à mesure, avec une approche descendante. Dans toute cette section, on considère une signature Σ abstraite, constante, dont les règles de bonne formation seront détaillées dans la section suivante.

Définition 3.9 (prédicats `wf`, `typ`)**.** Les règles de typage des environnements et des termes de $\mathrm{CIC_B}$ sont les suivantes.

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

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

$$\frac{\Gamma \vdash \quad s_1 \prec s_2}{\Gamma \vdash s_1 : s_2}\ \texttt{Typ\_srt} \qquad \frac{\Gamma \vdash \quad \Gamma(n) \text{ est de type } T}{\Gamma \vdash \natural n :\uparrow^{n+1} T}\ \texttt{Typ\_rel}$$

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

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

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma \vdash B : s_2 \quad s_1 \preccurlyeq s_3 \quad s_2 \preccurlyeq s_3}{\Gamma \vdash A * B : s_3}\ \texttt{Typ\_sum}$$

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

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

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

$$\frac{\Gamma \vdash \quad [\_ : T] \vdash \quad \vdash_0 c : T}{\Gamma \vdash c : T}\ \texttt{Typ\_cst0}$$

$$\frac{\Gamma[\_ : T] \vdash \quad \Gamma[\_ : U] \vdash \quad \Gamma \vdash t : T \quad \vdash_1 c(t) : T \Rightarrow U}{\Gamma \vdash c(t) : U}\ \texttt{Typ\_cst1}$$

Au besoin, on notera $\langle\Sigma\rangle\Gamma \vdash$ et $\langle\Sigma\rangle\Gamma \vdash t : T$ les jugements de typage complets, explicitant la signature.

Hiérarchie des types Le typage des environnements, ainsi que les quatre premières règles de typage des termes, sont typiques des PTS. Elles font appel à des relations `axiom` (notée $\prec$ en infixe dans `Typ_srt`) et `rule` généralement laissées abstraites dans la littérature traitant des PTS en général. Dans le cas de $\mathrm{CIC_B}$, ces relations sont fixées. La relation binaire `axiom` permet de typer les sortes :

Définition 3.10 (relation `axiom`)**.** La relation `axiom`, utilisée dans `Typ_srt`, est définie par les règles suivantes :

$$\frac{}{\mathsf{Prop} \prec \mathsf{Type}_n} \qquad \frac{}{\mathsf{Set} \prec \mathsf{Type}_n} \qquad \frac{n < m}{\mathsf{Type}_n \prec \mathsf{Type}_m}$$

`axiom` est une relation d'ordre bien fondée sur les sortes. On notera $\preccurlyeq$ en infixe la relation élargie à l'égalité. On remarquera que cette relation $\preccurlyeq$ sur les sortes ne coïncide pas exactement avec la relation $\leqslant$ de sous-typage vue à la définition 3.8, qui admet en plus $\mathsf{Prop} \leqslant \mathsf{Set}$. La relation élargie $\preccurlyeq$ est utilisée dans la règle `Typ_sum` de typage du type des paires non dépendantes. La relation ternaire `rule` permet quant à elle de typer les produits dépendants :

Définition 3.11 (relation `rule`)**.** La relation `rule`, utilisée dans `Typ_prd`, est définie par les règles suivantes :

$$\frac{s_0 \in \{\mathsf{Prop}, \mathsf{Set}\}}{\mathtt{rule}(s_0, s, s)} \qquad \frac{s_0 \in \{\mathsf{Prop}, \mathsf{Set}\}}{\mathtt{rule}(s, s_0, s_0)}$$

$$\frac{n \leqslant p \qquad m \leqslant p}{\mathtt{rule}(\mathsf{Type}_n, \mathsf{Type}_m, \mathsf{Type}_p)}$$

`rule` restreint les produits dépendants possibles. Prop et Set ont un statut particulier : si U est de type Prop, alors $\Pi x : \mathsf{Prop}.U$ est aussi de type Prop, donc x peut être instanciée par la formule dans laquelle elle apparaît. On dit que Prop est *imprédicatif*. Dans $\mathrm{CIC_B}$, Set est également imprédicatif. En revanche, si U est de type Type_n, alors $\Pi x : \mathsf{Type}_n.U$ est de type Type_{n+1} (le type de Type_n).

De plus, si $\mathtt{rule}(s_1, s_2, \mathsf{Prop})$, alors on a nécessairement $s_2 = \mathsf{Prop}$. Cette propriété est importante pour l'extraction : elle dit qu'une fonction dont le type est dans Prop ne peut renvoyer que des termes dont le type est dans Prop.

Les relations `axiom` et `rule`, ainsi que les règles `Typ_srt` et `Typ_prd` associées, exhibent une hiérarchie de sortes qui permet d'éviter des paradoxes. Elle est donc essentielle à la cohérence logique du système. La hiérarchie présentée ici correspond à celle de Coq 6.2. L'imprédicativité de Set étant incompatible avec certains axiomes raisonnables [20], elle a été retirée à partir de la version 8.0 : Set n'est plus qu'une sorte prédicative au même titre que les Type_i. Ce détail a une importance marginale dans le cadre de l'extraction.

Conversion La traditionnelle règle de conversion des PTS permet notamment d'effectuer des réductions dans les types. Ici, c'est `Typ_conv` qui joue ce rôle. Utilisant la relation de sous-typage, elle permet également d'élargir un type. La règle `Typ_conv_srt` est un reliquat de $\mathrm{CIC_A}$ et n'est pas théoriquement nécessaire dans $\mathrm{CIC_B}$, où existent une infinité de sortes toutes typées. Cependant, `Typ_conv_srt` permet par exemple de restreindre le système à un nombre fini N de sortes. Il serait alors envisageable de prouver la cohérence de ce sous-système dans un $\mathrm{CIC_B}$ à $M > N$ sortes sans contredire le théorème d'incomplétude de Gödel.

Opérateurs Les règles `Typ_cst0` et `Typ_cst1` typent les opérateurs constants et appliqués et font appel à des prédicats auxiliaires ne dépendant pas de `typ`. La règle `Typ_cst0` est relativement simple à exprimer car le terme c (un opérateur) que l'on cherche à typer ne possède pas de sous-terme. Elle fait appel à une relation binaire `mem_sign0` notée $\vdash_0$:

Définition 3.12 (relation `mem_sign0`)**.** Les opérateurs constants sont typés comme suit :

$$\frac{}{\vdash_0 \square : \mathcal{M}} \qquad \frac{}{\vdash_0 \mathcal{M} : \mathsf{Prop}} \qquad \frac{}{\vdash_0 \varepsilon : \mathcal{L}} \qquad \frac{}{\vdash_0 \mathcal{L} : \mathsf{Prop}}$$

En revanche, la règle `Typ_cst1` est plus complexe. Rappelons-la :

$$\frac{\Gamma[\_ : T] \vdash \quad \Gamma[\_ : U] \vdash \quad \Gamma \vdash t : T \quad \vdash_1 c(t) : T \Rightarrow U}{\Gamma \vdash c(t) : U}\ \texttt{Typ\_cst1}$$

Elle fait appel à un prédicat à quatre arguments — c, t, T et U — nommé `mem_sign1` et noté $\vdash_1 c(t) : T \Rightarrow U$. Les trois autres prémisses typent les termes t, T et U de sorte que `mem_sign1` n'a pas besoin de dépendre lui-même de `typ`. Afin de simplifier la présentation des règles constituant `mem_sign1`, remarquons que les constructions évoluées de $\mathrm{CIC_B}$ introduites dans le tableau de notations de la page 37 sont de la forme $c(t)$, où t dépend de sous-termes clairement identifiables $v_1, \ldots, v_n$. Le terme t a alors naturellement un type T contenant les types $V_1, \ldots, V_n$ de $v_1, \ldots, v_n$. Plutôt que de donner des règles de la forme :

$$\frac{}{\vdash_1 c(t) : T \Rightarrow U}$$

nous allons donner des règles de la forme :

$$\frac{\vdash v_1 : V_1 \qquad \cdots \qquad \vdash v_n : V_n}{\vdash_1 c(t) : U}$$

qui sont plus faciles à lire en tant que règles de typage. Intuitivement, les pseudo-prémisses $\vdash v_1 : V_1$ à $\vdash v_n : V_n$ correspondent à la prémisse $\Gamma \vdash t : T$ de `Typ_cst1` décomposée mais, formellement, $\vdash_1$ ne dépend pas de $\vdash$.

Définition 3.13 (prédicat `mem_sign1`)**.** Les opérateurs appliqués sont typés par les règles suivantes.

$$\frac{\vdash h : \mathcal{M} \qquad \vdash q : \mathcal{L}}{\vdash_1 h :: q : \mathcal{L}}$$

$$\frac{\vdash t : A * B}{\vdash_1 \pi_1(t) : A} \qquad \frac{\vdash t : A * B}{\vdash_1 \pi_2(t) : B}$$

$$\frac{\vdash f : \Pi x : A.B \qquad \vdash m : A}{\vdash_1 f\ m : B\,\{\natural 0 \leftarrow m\}}$$

$$\frac{\Sigma(C) = [\langle C, P, t, T\rangle \mid \langle C, P, T\rangle : \texttt{constant\_obj}] \qquad \vdash p : P}{\vdash_1 \texttt{Const}(C)(p) : T\,\{\natural 0 \leftarrow p\}}\ \texttt{Mem\_cst}$$

$$\frac{\vdash m : s \qquad m \equiv_{\mathrm{B}} T_1 * (\Pi x_1 : T_1.\cdots(T_n * \Pi x_n : T_n.\mathcal{L})) \qquad \text{pour tous } i \text{ et } j,\ x_i = x_j \text{ implique } i = j}{\vdash_1 (\!|\, \vec{x} : m \,|\!) : s}$$

$$\frac{\vdash (\!|\, \vec{x} : \vec{T} \,|\!) : s \qquad \vdash \{\vec{t}\} : T_1 * (T_2\,\{\natural 0 \leftarrow t_0\} * \cdots (T_n\,\{\natural 0 \leftarrow t_0\}\cdots\{\natural 0 \leftarrow t_{n-1}\} * \mathcal{L}))}{\vdash_1 (\!|\, \vec{x} : \vec{T} := \vec{t} \,|\!) : (\!|\, \vec{x} : \vec{T} \,|\!)}$$

$$\frac{y = x_i \qquad \vdash t : (\!|\, \vec{x} : \vec{T} \,|\!)}{\vdash_1 \texttt{Field}(y)(t) : T_i}$$

$$\frac{\vdash \{\vec{T}\} : s \qquad \vdash \lambda m : \mathcal{M}.\lambda f : \vec{T}^{-m}.\{\vec{t}\} : \Pi m : \mathcal{M}.\Pi f : \vec{T}^{-m}.\vec{T}^{+m}}{\vdash_1 \mathsf{Fix}\{\vec{t} : \vec{T}\} : \{\vec{T}\}}\ \texttt{Mem\_fix}$$

$$\frac{\begin{array}{c}\vdash t : \mathsf{Ind}\{I \mid p, a\}\langle n, l\rangle \qquad \beta \dashv I \Rightarrow s \\ \vdash Q : \Pi a : A.\Pi y : \mathsf{Ind}\{I \mid p, a\}\langle 0, \varepsilon\rangle.s \qquad Q\langle l\rangle \dashv \vec{x} : \vec{B} \\ \Sigma(I) = \langle I, P, A, r, s, \vec{c}\rangle : \texttt{one\_ind} \qquad \vdash \{\vec{b}\} : \{\vec{B}\}\end{array}}{\vdash_1 \mathsf{Match}_\beta\{t \Rightarrow \vec{x} \mid Q \Rightarrow \vec{b}\} : Q\ a\ t}\ \texttt{Mem\_case}$$

$$\frac{\begin{array}{c}\Sigma(I) = \langle I, P, A, r, s, \vec{c}\rangle : \texttt{one\_ind} \\ \vdash p : P \qquad \vdash a : A\,\{\natural 0 \leftarrow p\} \qquad \vdash l : \mathcal{L}\end{array}}{\vdash_1 \mathsf{Ind}\{I \mid p, a\}\langle n, l\rangle : r}\ \texttt{Mem\_mutind}$$

$$\frac{\begin{array}{c}\Sigma(I) = \langle I, P, A, r, s, \vec{c}\rangle : \texttt{one\_ind} \\ \Sigma(C) = \langle C, T, a\rangle : \texttt{constructor} \\ C \in \vec{c} \qquad \vdash p : P \qquad \vdash t : T\,\{\natural 0 \leftarrow p\}\end{array}}{\vdash_1 \mathsf{Constr}\{C \mid p, t\} : \mathsf{Ind}\{I \mid p, a\,\{\natural 1 \leftarrow p\}\,\{\natural 0 \leftarrow t\}\}\langle 0, \varepsilon\rangle}\ \texttt{Mem\_constr}$$

Les règles `Mem_fix` et `Mem_case` font appel aux notations $\vec{T}^{\pm m}$ et $Q\langle l\rangle \dashv \vec{x} : \vec{B}$, vues à la section 3.1.3. La règle `Mem_case` utilise également une nouvelle notion $\beta \dashv I \Rightarrow s$ que nous allons maintenant expliquer.

Typage des filtrages Tout comme pour les produits dépendants, il est nécessaire (d'un point de vue logique) d'imposer des contraintes sur les sortes mises en jeu lors d'un filtrage, à savoir la sorte de l'argument des constructeurs, de l'inductif lui-même, et celle du prédicat de retour. La prémisse $\beta \dashv I \Rightarrow s$ de `Mem_case` fait appel à un prédicat ternaire `elim_sort` mettant en relation la sorte des constructeurs, la sorte de l'inductif et la sorte du prédicat de retour. `elim_sort` joue un rôle similaire à celui de `rule`.

Définition 3.14 (relation `elim_sort`)**.** La relation `elim_sort` est définie par les règles suivantes :

$$\frac{}{\texttt{elim\_sort}(s, \mathsf{Prop}, \mathsf{Prop})}$$

$$\frac{s_0 \in \{\mathsf{Prop}, \mathsf{Set}\}}{\texttt{elim\_sort}(s_0, \mathsf{Set}, s)} \qquad \frac{s_0 \in \{\mathsf{Prop}, \mathsf{Set}\}}{\texttt{elim\_sort}(\mathsf{Type}_n, \mathsf{Set}, s_0)}$$

$$\frac{s_0 \in \{\mathsf{Prop}, \mathsf{Set}\}}{\texttt{elim\_sort}(s_0, \mathsf{Type}_n, s)} \qquad \frac{n \leqslant m}{\texttt{elim\_sort}(\mathsf{Type}_n, \mathsf{Type}_m, s)}$$

La première règle implique que si $\texttt{elim\_sort}(s_1, \mathsf{Prop}, s_2)$, alors $s_2 = \mathsf{Prop}$. En d'autres termes, le filtrage d'un terme dont le type est dans Prop ne peut construire que des termes dont le type est dans Prop, une autre propriété essentielle pour l'extraction. Les autres règles vont de pair avec `rule` et ont des justifications logiques qui n'affectent pas l'extraction.

En plus des filtrages satisfaisant la relation `elim_sort`, les filtrages sur des inductifs vides ou ne possédant qu'un seul constructeur dont l'argument est dans Prop, et construisant quelque chose hors de Prop, sont autorisés. Cette exception n'était pas présente dans $\mathrm{CIC_A}$, mais est présente dans Coq et très utilisée en pratique, notamment pour la réécriture et la récursion générale. Nous avons donc adapté le système en conséquence, et c'est la principale différence entre $\mathrm{CIC_A}$ et $\mathrm{CIC_B}$.

Nous avons choisi de rendre explicites au niveau de la syntaxe les cas particuliers de filtrage. Pour cela, le paramètre β de l'opérateur de filtrage permet de distinguer les deux cas de restriction (régulier ou spécial). Si $\Sigma(I) = \langle I, P, A, r, s, \vec{c}\rangle$, on notera $\beta \dashv I \Rightarrow s'$ si on est dans l'un des cas suivants :

- $\beta = \mathsf{r}$ et $\texttt{elim\_sort}(s, r, s')$;
- $\beta = \mathsf{s}$ et $|\vec{c}| = 0$;
- $\beta = \mathsf{s}$, $|\vec{c}| = 1$ et $s = \mathsf{Prop}$.

Ce paramètre est inexistant dans Coq, mais nous l'avons ajouté dans $\mathrm{CIC_B}$ afin de simplifier la formulation des règles de réduction dans le langage cible de notre extraction (définition 3.25). Les particularités de la règle de typage associée sont néanmoins implantées dans Coq, et ce paramètre pourrait être inféré dans une phase de pré-typage.

Termes logiques Le typage nous permet maintenant de définir précisément la notion de terme logique.

Définition 3.15. Un terme t (considéré dans un environnement Γ et une signature Σ) est dit *logique* s'il existe T tel que $\langle\Sigma\rangle\Gamma \vdash t : T$ et $\langle\Sigma\rangle\Gamma \vdash T : \mathsf{Prop}$. Il est dit *informatif* sinon.

3.1.5 Typage des signatures

Dans le cas d'une signature vide, les règles de réduction `delta_glob`, `iota_case` et `iota_fix`, et les règles de typage

`Mem_cst`, `Mem_fix`, `Mem_case`, `Mem_mutind`, et `Mem_constr`

sont inapplicables. On obtient néanmoins un système logique intéressant qui correspond approximativement à ECC [34].

$\mathrm{CIC_B}$ dispose, via la signature, d'un mécanisme générique d'ajout de nouvelles constructions au langage, avec des règles de réduction et de typage associées. Les constantes (accessibles depuis les termes via l'opérateur `Const`) jouent un rôle de structuration et de factorisation de code, et leur déclaration ne demande pas de contrainte particulière.

Définition 3.16 (prédicat `wf_cst`)**.**
- La définition $o = \langle C, P, t, T\rangle$ est bien formée dans Σ si $\langle\Sigma\rangle[\_ : P][\_ \doteq t : T] \vdash$ et $C \notin \Sigma$.
- L'axiome $o = \langle C, P, T\rangle$ est bien formé dans Σ si $\langle\Sigma\rangle[\_ : P][\_ : T] \vdash$ et $C \notin \Sigma$.

Dans les deux cas, on notera $\texttt{wf\_cst}(\Sigma, o)$.

En revanche, les inductifs (opérateurs `MutInd`, `MutConstr`, `MutCase` et `Fixp`) doivent respecter des contraintes plus strictes afin de garantir la cohérence du système.

Définition 3.17 (prédicats `wf_constructor`, `constr_pos`)**.** Soient Σ une signature, $\vec{J}$ une liste de noms, $i = \langle I, P, A, r, s, d\rangle$ une déclaration d'inductif (type `one_ind`) avec $I \in \vec{J}$, $c = \langle C, T, a\rangle$ une déclaration de constructeur (type `constructor`). On dit que c est bien formée si :

1. $[x : P] \vdash T : s$;
2. $[x : P][y : A] \vdash a : A$;
3. c satisfait la condition de positivité ;

4. C n'apparaît pas dans Σ.

Cette condition sera notée $\texttt{wf\_constructor}(\Sigma, \vec{J}, i, c)$.

On dit que c satisfait la *condition de positivité* si aucun J_j n'apparaît dans a, et s'ils n'apparaissent dans T qu'en position positive. Les positions positives sont à droite des produits dépendants ($\Pi x : T_1.\_$), à droite ou à gauche des types de paire ($\_ * \_$), et à toutes les positions des types d'enregistrement ($(\!|\, \vec{x} : \_ \,|\!)$). De plus, chaque occurrence doit être de la forme $\mathsf{Ind}\{J_k \mid x, t\}\langle n, \varepsilon\rangle$, où x est la variable du premier point ci-dessus (libre et représentée par $\natural 0$ dans la déclaration brute), et t un terme ne contenant aucun J_j.

La condition de positivité [1] ci-dessus est plus restrictive que celle actuellement implantée dans Coq.

On dit parfois qu'un paramètre est *récursivement uniforme* lorsqu'il est le même dans toutes les occurrences récursives (condition « chaque occurrence doit être de la forme $\mathsf{Ind}\{J_k \mid x, t\}\langle n, \varepsilon\rangle$ » dans la définition précédente). Dans $\mathrm{CIC_B}$ — et dans Coq à l'époque du développement de B. Barras — tous les paramètres d'inductif sont récursivement uniformes, mais ce n'est plus le cas dans Coq aujourd'hui : par exemple, le type `Acc` de la section 2.1.3 ne l'est pas. Les conséquences logiques de la version étendue implantée dans Coq n'étant pas claires dans le cadre de $\mathrm{CIC_B}$, nous avons préféré ne pas y toucher.

Cette condition interdit également les inductifs emboîtés, où un inductif J est utilisé dans le paramètre d'un autre inductif I dans le type d'un constructeur de J, comme dans cet exemple de définition d'arbre :

```
Inductive tree (A : Type) := node : list (tree A) -> tree A.
```

Les inductifs emboîtés ne sont pas admis dans $\mathrm{CIC_B}$ à cause de la condition de garde des points fixes à base de marques (voir la section 7.7.1 de [5]).

Définition 3.18 (prédicats `wf_arity`, `wf_constr`)**.** Soit $\vec{i}$ une déclaration inductive ne contenant aucun inductif abstrait. On note i_k la déclaration $\langle I_k, P_k, A_k, r_k, s_k, \vec{c_k}\rangle$ pour le k-ième inductif, et $c_{k,l}$ la déclaration $\langle C_{k,l}, T_{k,l}, a_{k,l}\rangle$ pour le l-ième constructeur du k-ième inductif. La déclaration $\vec{i}$ est bien formée dans Σ si :

1. les noms d'inductifs sont deux à deux distincts et, pour tout k,

$$\langle\Sigma\rangle[\_ : P_k][\_ : A_k][\_ : r_k] \vdash \qquad \text{et} \qquad I_k \notin \Sigma$$

   (condition notée $\texttt{wf\_arity}(\Sigma, \vec{i})$) ;

2. les noms de constructeurs sont deux à deux distincts et, pour tous k et l,

$$\texttt{wf\_constructor}(\Sigma[\vec{i^\star}], \vec{I}, i_k, c_{k,l})$$

   (condition notée $\texttt{wf\_constr}(\Sigma[\vec{i^\star}], \vec{i})$).

1. qui est qualifiée de *stricte* dans la littérature

On rappelle que $\vec{i}^{\star}$ est la déclaration $\vec{i}$ où tous les inductifs sont rendus abstraits (oubli des constructeurs).

Définition 3.19 (prédicat `wf_sign`)**.** Les règles de typage des signatures sont les suivantes :

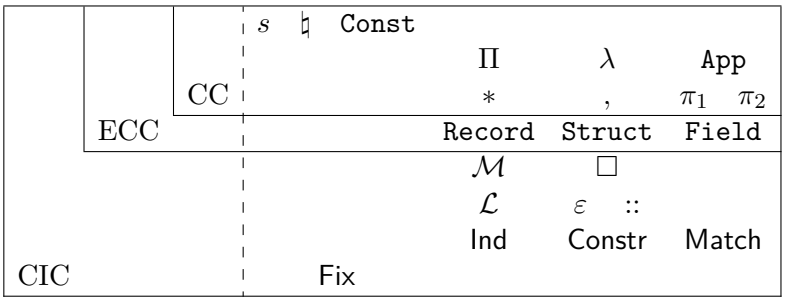
$$\frac{}{\langle[]\rangle \vdash} \qquad \frac{\langle\Sigma\rangle \vdash \qquad \texttt{wf\_cst}(\Sigma, o)}{\langle\Sigma[o]\rangle \vdash}$$

$$\frac{\langle\Sigma\rangle \vdash \qquad \texttt{wf\_arity}(\Sigma, \vec{i}) \qquad \texttt{wf\_constr}(\Sigma[\vec{i}^{\star}], \vec{i})}{\langle\Sigma[\vec{i}]\rangle \vdash}$$

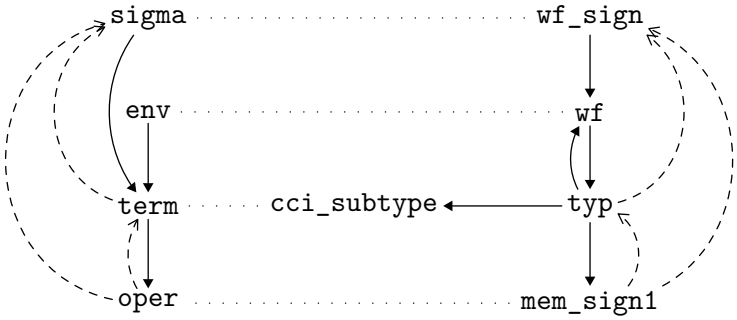
Dans la suite, on supposera toujours travailler avec des signatures bien typées et, parfois, on omettra même toute référence à la signature.

3.1.6 Récapitulatif

On peut classifier les 22 constructions syntaxiques de $\mathrm{CIC_B}$ par « ère » logique (CC [13], ECC [34], CIC [47]), concept (produit dépendant, paire…) et nature (type, constructeur, destructeur) :



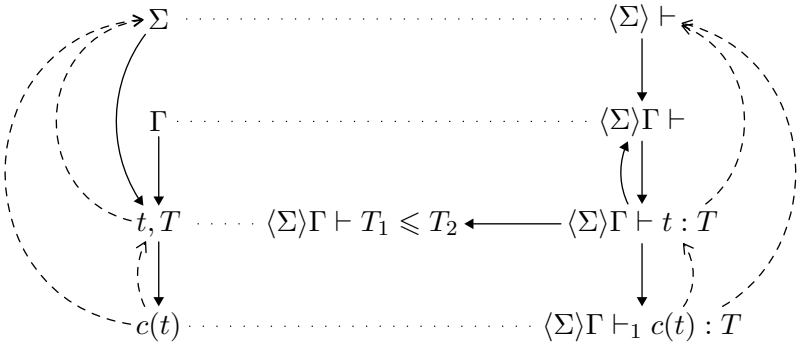
Les relations qui lient les différentes définitions peuvent être résumées par le diagramme suivant :



La colonne de gauche représente les différents types syntaxiques, et la colonne de droite les jugements de typage correspondants (nous ne considérons que les opérateurs appliqués). Le sous-typage (qui inclut la réduction) est entre syntaxe et typage.

Les flèches pleines représentent les dépendances « brutes » : la définition des termes dépend des opérateurs, etc. Les flèches en pointillés représentent des dépendances indirectes : un opérateur peut faire référence à des noms déclarés dans la signature, et ne peut avoir de sens qu'appliqué à un terme d'une certaine forme, etc.

Des notations ont été introduites pour tous ces concepts :



3.1.7 Propriétés

B. Barras a prouvé en Coq de nombreuses propriétés sur toutes les définitions vues jusqu'à présent, comme le fait que le sous-typage est un pré-ordre qui est stable par affaiblissement, substitution, et sous-typage dans un type de l'environnement, ou encore que $\mathrm{CIC_B}$ possède les propriétés habituelles d'affaiblissement, de substitution, et de préservation du typage par réduction (en anglais, *subject reduction*). De plus, tous les jugements de typage sont décidables. Néanmoins, ces résultats de décidabilité reposent sur un axiome de terminaison :

Axiome 3.20 (`cci_is_strongly_normalizing_axiom`)**.** *Si $\Gamma \vdash t : T$, alors t normalise fortement.*

Cet axiome n'est pas directement prouvable en Coq car il implique la cohérence logique de $\mathrm{CIC_B}$, que nous pensons équivalent à Coq. Il est toutefois envisageable de reposer sur des axiomes plus conventionnels, comme ceux de la théorie des ensembles [6]. En pratique, on se sert surtout de la mise en forme normale de tête faible.

La présentation à la Church du système (toutes les variables sont annotées par leur type dans les lieurs) donne aussi un résultat d'inférence (non comparable avec le prétypage de Coq) et de principalité :

Lemme 3.21 (`inf_ppal_type`)**.** *Soit t un terme de* $\mathrm{CIC_B}$*. L'un des deux énoncés suivants est vrai :*

– *t admet un type principal T, c'est-à-dire* $\Gamma \vdash t : T$ *et pour tout* T' *tel que* $\Gamma \vdash t : T'$, *on a* $\Gamma \vdash T \leqslant T'$;
– *t n'est pas typable : pour tout T, on n'a pas* $\Gamma \vdash t : T$.

Il s'agit là d'un des principaux résultats de B. Barras. L'extrait informel de ce lemme et d'autres similaires produit un noyau comparable à celui de Coq, utilisé au cœur de Bcoq.

Nous avons utilisé intensivement les résultats de B. Barras dans la formalisation de l'extraction faite à la section 3.2. Dans nos preuves en Coq, nous avons admis certains sous-buts qui correspondent à des cas particuliers du lemme suivant :

Lemme 3.22. *Soit t un terme clos (localement et globalement) et en forme normale de tête faible tel que* $\Gamma \vdash t : T$. *On suppose de plus que tous les points fixes apparaissant dans t sont totalement appliqués (c'est-à-dire apparaissent toujours sous la forme* $\pi_1(\pi_2^n(\mathsf{Fix}\{\vec{f} : \vec{T}\}))\ p\ t$*). On a :*
– *si T est de la forme* $\Pi x : U_1.U_2$, *alors il existe y,* U_1' *et u tels que* $t = \lambda y : U_1'.u$;
– *si T est de la forme* $U_1 * U_2$, *alors il existe* u_1 *et* u_2 *tels que* $t = (u_1, u_2)$;
– *si T est de la forme* $\mathsf{Ind}\{I \mid p, a\}\langle n, l\rangle$, *alors il existe* a' *et un constructeur C de I tel que* $t = \mathsf{Constr}\{C \mid p, a'\}$;
– *si T est de la forme* $(\!|\, \vec{x} : \vec{U} \,|\!)$, *alors il existe* $\vec{U'}$ *et* $\vec{u}$ *tels que* $t = (\!|\, \vec{x} : \vec{U'} := \vec{u} \,|\!)$.

Preuve informelle. Ce résultat peut se prouver par analyse de cas sur t puis inversion du jugement de typage $\Gamma \vdash t : T$. Tous les cas gênants vont à l'encontre de l'hypothèse que t est clos ou que t est en forme normale. □

Ce lemme est inexistant dans le développement original de B. Barras, car inutile. Cependant, nous en avons besoin dans nos preuves concernant l'extraction. On peut la considérer comme une adaptation du lemme 3 de [33] au cadre de $\mathrm{CIC_B}$, mais elle n'a pas été facile à réaliser. La condition d'application totale des points fixes, sans laquelle le premier cas ne serait pas vérifié, n'est pas facile à exprimer dans le développement Coq sous sa forme présente. Aussi ce lemme n'a-t-il pas été prouvé (ni même énoncé) formellement. Nous avons essayé de le faire en imposant la condition directement au niveau de la syntaxe (en forçant les points fixes à être toujours appliqués), et la preuve semble faisable. Toutefois, il s'agit d'un changement très intrusif, et nous n'avons pas adapté le développement dans son intégralité, faute de temps.

3.2 Formalisation de l'extraction

Nous allons maintenant présenter notre formalisation de l'extraction proprement dite. Le langage cible de notre extraction, que nous appelons $\mathrm{CIC_{B'}}$, est essentiellement $\mathrm{CIC_B}$ sans types et avec une relation de réduction modifiée. Nous définissons dans un premier temps une relation d'extraction, reliant les termes de $\mathrm{CIC_B}$ à des extraits potentiels,

et donnons des propriétés de correction exprimées à l'aide de cette relation. Puis nous définissons une fonction d'extraction, et discutons de son intégration à Bcoq.

Cette section s'inspire de la section 2.3 de [33], et notre contribution est l'adaptation à $\mathrm{CIC_B}$, ainsi que les preuves en Coq des différents résultats.

3.2.1 Relation d'extraction

Dans le développement Coq, nous avons choisi de conserver le même type de données pour les termes de $\mathrm{CIC_{B'}}$, c'est-à-dire `term`. Cependant, seuls les termes images de l'extraction sont considérés. L'extraction décrite ici remplace tous les types et les sous-termes logiques par un substitut $\square$ (on dira alors qu'ils sont *effacés*). Comme expliqué dans [33], il est nécessaire dans un premier temps de définir une relation d'extraction non fonctionnelle.

Définition 3.23 (prédicat `Pe`)**.** La relation d'extraction $t \to_{\mathcal{E}} t'$, qui dépend implicitement d'un environnement Γ, est définie par les règles suivantes :

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

$$\frac{t \in \texttt{sort}}{t \to_{\mathcal{E}} \square}\ \texttt{Pe\_Srt} \qquad \frac{}{t_1 * t_2 \to_{\mathcal{E}} \square}\ \texttt{Pe\_Pair} \qquad \frac{}{\Pi x : T.t \to_{\mathcal{E}} \square}\ \texttt{Pe\_Prd}$$

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

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

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

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

$$\frac{t \to_{\mathcal{E}} t' \quad b \to_{\mathcal{E}} b'}{\mathsf{Match}_\beta\{t \Rightarrow \vec{x} \mid P \Rightarrow b\} \to_{\mathcal{E}} \mathsf{Match}_\beta\{t' \Rightarrow \vec{x} \mid \square \Rightarrow b'\}}\ \texttt{Pe\_Case}$$

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

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

On dira alors que t s'*extrait* (partiellement) vers t' dans le contexte Γ. On notera $\Gamma \vdash t \to_{\mathcal{E}} t'$ pour rendre l'environnement explicite.

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 `Pe` 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. L'extraction modélisée par `Pe` peut être partielle : tous les sous-termes logiques ne sont pas forcément effacés (l'application de la règle `Pe_Prop` est facultative).

`Pe` peut s'appliquer directement aux encodages de listes dans les termes : ainsi, dans `Pe_Case`, b et b' sont des listes, mais on ne considère que leurs encodages. Cependant, l'encodage n'est pas idéal vis-à-vis de l'extraction : l'extrait d'une liste n'est pas forcément une liste de même taille. En effet, la liste extraite peut être tronquée dès que tous les éléments restants sont logiques.

Notons qu'il existe une ambigüité concernant le symbole $\square$ dans $\mathrm{CIC}_{\mathrm{B}'}$: il peut s'agir soit de la marque concrète, soit du substitut introduit par `Pe`. L'ambigüité est levée dans la plupart des cas grâce au typage de $\mathrm{CIC}_{\mathrm{B}}$, mais elle subsiste, par exemple, dans le jugement $\square \to_{\mathcal{E}} \square$, qui peut être produit à la fois par `Pe_Prop` et par `Pe_Cst0`.

3.2.2 Réduction

Les principales propriétés de l'extraction qui ont été formalisées sont des résultats de simulation : les réductions dans $\mathrm{CIC}_{\mathrm{B}}$ et dans $\mathrm{CIC}_{\mathrm{B}'}$ peuvent être mises en correspondance. En d'autres termes, la valeur précise des types et des sous-termes logiques n'est pas pertinente pour le calcul.

Dans le cadre de cette étude, nous ne nous intéressons qu'aux termes clos. Par conséquent, l'opérateur de passage au contexte est restreint aux contextes sans lieurs (la réduction est dite *faible*). La réduction faible est d'ailleurs souhaitable dans le cadre d'application de l'extraction : $\mathrm{CIC}_{\mathrm{B}'}$ se veut être une première étape vers un langage de programmation compilable et exécutable efficacement, où la réduction est traditionnellement faible. De plus, nous avons vu à la section 2.7.2 que la réduction prématurée de termes ouverts pouvait causer des erreurs à l'exécution.

Définition 3.24 (opération `wctxt`)**.** Soit $\to$ une règle de réduction. Les règles suivantes définissent la *clôture par contexte faible* $\to_{\mathcal{W}}$ de $\to$:

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

$$\frac{T \to_{\mathcal{W}} T'}{\lambda x : T.M \to_{\mathcal{W}} \lambda x : T'.M}\ \texttt{WCtx\_lam\_l}$$

$$\frac{T \to_{\mathcal{W}} T'}{\Pi x : T.U \to_{\mathcal{W}} \Pi x : T'.U}\ \texttt{WCtx\_prd\_l}$$

$$\frac{A \to_{\mathcal{W}} A'}{A * B \to_{\mathcal{W}} A' * B}\ \texttt{WCtx\_sum\_l} \qquad \frac{B \to_{\mathcal{W}} B'}{A * B \to_{\mathcal{W}} A * B'}\ \texttt{WCtx\_sum\_r}$$

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

Bien que leur valeur précise ne soit pas pertinente, il est possible que des termes logiques interviennent dans des réductions de termes informatifs, en particulier dans les cas suivants :

- l'argument inductif d'un point fixe informatif peut être logique et s'extraire vers $\square$, enlevant tout espoir de déclenchement de la règle `iota_fix`. C'est le cas quand on utilise la récursion bien fondée (type `Acc`) pour définir une fonction récursive ;
- le filtrage de $\square$ peut se retrouver en tête dans les cas spéciaux du filtrage ($\beta = \mathtt{s}$) : la règle `iota_case` ne peut alors plus s'appliquer. C'est le cas quand on utilise la réécriture dans les types (type `eq`) ;
- lorsque t est de type $A * \mathcal{M}$, il peut être logique ou informatif selon la valeur de A, et il est possible de construire un terme informatif utilisant $\pi_1(t)$, nécessitant vraiment la réduction de cette projection lors du calcul. Or lorsque $A = \mathcal{M}$, t peut être directement extrait vers $\square$, pouvant mener la réduction à se retrouver bloquée face à un $\pi_1(\square)$.

De plus, la définition de `Pe` n'impose pas le remplacement par $\square$ dès que possible, laissant par exemple la possibilité de produire directement $\pi_1(\square)$ (caractère partiel de la relation d'extraction). Tous ces points montrent la nécessité d'ajouter des règles de réduction à celles de $\mathrm{CIC_B}$.

Définition 3.25 (relation `dummy_redn_term`, $\to_{\mathrm{B}'}$)**.** La relation `dummy_redn_term` est l'union de `redn_term` et des règles suivantes :

$$\frac{}{\pi_1\ \square \to \square}\ \texttt{dummy\_proj1} \qquad \frac{}{\pi_2\ \square \to \square}\ \texttt{dummy\_proj2}$$

$$\frac{}{\texttt{Field}(y)\Box \to \Box}\ \texttt{dummy\_projr} \qquad \frac{}{\Box\ N \to \Box}\ \texttt{dummy\_beta}$$

$$\frac{}{\mathsf{Match_r}\{\Box \Rightarrow \vec{x} \mid P \Rightarrow \vec{b}\} \to \Box}\ \texttt{dummy\_iota\_case}$$

$$\frac{}{\mathsf{Match_s}\{\Box \Rightarrow x \mid P \Rightarrow b\} \to b\ \Box}\ \texttt{dummy\_iota\_case\_pi}$$

$$\frac{F = \mathsf{Fix}\{\vec{f} : \Box\} = \texttt{Fixp}((\Box, F_0))}{\pi_1(\pi_2^n F)\ p\ \Box \to \pi_1(\pi_2^n(F_0\ \Box\ F))\ p\ \Box}\ \texttt{dummy\_iota\_fix}$$

Chacune des règles supplémentaires correspond à une règle de `redn_term` où un redex « disparaît » à cause de l'effacement d'un sous-terme. On peut distinguer deux catégories de règles :

- les cinq premières règles correspondent à des « redex dégénérés », c'est-à-dire ne pouvant plus rien construire d'informatif. Ces règles ne font que propager l'effacement des sous-termes ;
- les deux dernières règles correspondent à des « redex bureaucratiques », qui servent à extirper des termes informatifs qui étaient protégés par un argument logique dans $\mathrm{CIC_B}$.

3.2.3 Propriétés

Nous présentons ici les principaux résultats qui ont été prouvés en Coq dans le cadre de cette thèse. Par souci de simplicité, les énoncés sont donnés avec des variables nommées, et toutes les opérations sur les variables de de Bruijn sont omises.

Comme la relation de typage, la relation d'extraction vérifie des lemmes d'affaiblissement et de substitution. Les trois prochains lemmes sont directement inspirés du typage.

Le lemme suivant permet d'ajouter une déclaration (d) n'importe où dans un environnment ($\Gamma\Gamma'$) :

Lemme 3.26 (`Pe_weak`)**.** *La règle suivante est admissible :*

$$\frac{\langle\Sigma\rangle \vdash \quad \langle\Sigma\rangle\Gamma d \vdash \quad \Gamma\Gamma' \vdash t \to_{\mathcal{E}} t'}{\Gamma d \Gamma' \vdash t \to_{\mathcal{E}} t'}$$

Preuve informelle. Par induction sur $\Gamma\Gamma' \vdash t \to_{\mathcal{E}} t'$. Le cas `Pe_Prop` est conséquence de l'affaiblissement du typage. Tous les autres cas sont immédiats. □

Ce lemme se généralise aux extensions arbitraires de l'environnement :

Lemme 3.27 (`Pe_thinning`)**.** *La règle suivante est admissible :*

$$\frac{\langle\Sigma\rangle \vdash \quad \langle\Sigma\rangle\Gamma\Gamma' \vdash \quad \Gamma \vdash t \to_{\mathcal{E}} t'}{\Gamma\Gamma' \vdash t \to_{\mathcal{E}} t'}$$

Preuve informelle. Par induction sur Γ', en utilisant le lemme précédent. □

Les deux lemmes précédents formulés avec des variables nommées laissent penser que les manipulations dans l'environnement n'affectent pas les termes t et t', mais il ne faut pas oublier que la représentation de de Bruijn implique des renumérotations à chaque ajout dans l'environnement. Par exemple, l'énoncé Coq de `Pe_thinning` est :

```
forall sg t' t e f n,
  trunc n f e -> sg -- |- f -> Pe e t t' ->
    Pe f ^(n)t ^(n)t'.
```

La relation d'extraction est également préservée après substitution :

Lemme 3.28 (`Pe_sub`)**.** *La règle suivante est admissible :*

$$\frac{\langle\Sigma\rangle \vdash \quad \Gamma d\Gamma' \vdash t : T \quad \Gamma d\Gamma' \vdash t \to_{\mathcal{E}} t' \quad \Gamma \vdash u : U \quad \Gamma \vdash u \to_{\mathcal{E}} u' \quad d \in \{[x : U], [x \doteq u : U]\}}{\Gamma\Gamma'[x := u] \vdash t[x := u] \to_{\mathcal{E}} t'[x := u']}$$

Preuve informelle. Par induction sur $\Gamma d\Gamma' \vdash t \to_{\mathcal{E}} t'$. Le cas `Pe_Prop` est conséquence du lemme de substitution du typage. Le cas `Pe_Rel` lorsque la variable est différente de x se prouve en utilisant le lemme précédent. □

La formulation et la preuve en Coq des deux théorèmes suivants sont les principaux résultats de ce chapitre : ils expriment l'équivalence entre un terme $\mathrm{CIC_B}$ et ses extraits. D'une part, les réductions dans $\mathrm{CIC_{B'}}$ peuvent être simulées dans $\mathrm{CIC_B}$:

Théorème 3.29 (`Pe_redn_correct`)**.** *Soit t un terme $\mathrm{CIC_B}$ clos et bien typé. Si $t \to_{\mathcal{E}} t'$ et $t' \to_{\mathrm{B'}} u'$, alors il existe u tel que $u \to_{\mathcal{E}} u'$ et $t \to^{+}_{\mathrm{B}\mathcal{W}} u$.*

La formulation en Coq de ce théorème est :

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

Preuve informelle. Par analyse de cas sur $t' \to_{\mathrm{B'}} u'$.

- Toutes les règles issues de `redn_term` imposent une forme particulière à t' et à u' et, par inversion de $t \to_{\mathcal{E}} t'$, imposent une forme particulière à t permettant de déclencher la même réduction dans $\mathrm{CIC_B}$. Le t' obtenu s'extrait alors vers u' en utilisant la même règle que pour $t \to_{\mathcal{E}} t'$.
- Dans le cas d'un redex dégénéré (où on a nécessairement $u' = \square$), on réduit la pré-image du $\square$ apparaissant à gauche de la règle de réduction (cela peut provoquer des réductions arbitraires sous un contexte faible) ; le lemme 3.22 permet alors d'obtenir la forme de cette pré-image, puis de déclencher une réduction de t à l'aide d'une règle de `redn_term`. Le typage est tel que le t' obtenu est logique et peut être extrait via `Pe_Prop` vers $\square$.
- Le cas d'un redex bureaucratique est similaire à celui d'un redex dégénéré, mais diffère par le fait que u' ne vaut pas nécessairement $\square$. Le terme t' obtenu s'extrait alors vers u' en utilisant la même règle que pour $t \to_{\mathcal{E}} t'$.

$\square$

D'autre part, les réductions dans $\mathrm{CIC_B}$ peuvent être simulées dans $\mathrm{CIC_{B'}}$:

Théorème 3.30 (`Pe_redn_correct_source`)**.** *Soit t un terme $\mathrm{CIC_B}$ clos et bien typé. Si $t \to_{\mathcal{E}} t'$ et $t \to_{\mathrm{B}} u$, alors il existe u' tel que $u \to_{\mathcal{E}} u'$ et $t' \to_{\mathrm{B'}}^{*} u'$.*

La formulation en Coq de ce théorème est :

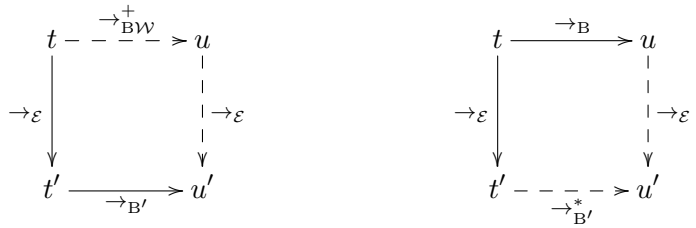
```
forall sg, wf_sign sg -> forall e t u t' T,
  closed 0 t -> gclosed t ->
  sg -- e |- t : T ->
  Pe e t t' ->
  redn_term e t u ->
    exists u', R_rt dummy_redn_term e t' u' /\ Pe e u u'
```

Preuve informelle. Par analyse de cas sur $t \to_{\mathrm{B}} u$, puis inversion de $t \to_{\mathcal{E}} t'$. On peut distinguer trois situations :

- t s'extrait vers $\square$. t est donc logique, et par préservation du typage par réduction, u est aussi logique et peut s'extraire vers $\square$. Aucune réduction de t' n'est nécessaire ;
- un sous-terme de t nécessaire à la réduction s'extrait vers $\square$. Par exemple, $t = \pi_1(t_0)$, et t_0 est logique et s'extrait vers $\square$. Dans ce cas, on utilise une des règles qui ont été ajoutées dans `dummy_redn_term` pour réduire t' ;
- t est suffisamment conservé par l'extraction pour que la même règle de réduction s'applique à t'.

$\square$

Ces deux théorèmes peuvent être résumés par les diagrammes commutatifs suivants :



La formulation et la preuve en Coq du théorème 3.29 (théorème 2 de [33]) étaient les principaux résultats de nos travaux de master [22, 23]. Toutefois, la formalisation était incomplète à l'époque. En particulier, le lemme 3.22 n'avait pas été clairement identifié. La réciproque 3.30 (théorème 4 de [33]), quant à elle, est totalement nouvelle dans cette thèse, et c'est elle qui a motivé certains changements par rapport au système présenté dans [23]. En particulier, la preuve originale de P. Letouzey supposait implicitement que la règle `Pe_Prop` était toujours appliquée si possible, biais qui a été mis en évidence par la formalisation. C'est en corrigeant cette preuve que nous est venue l'idée d'ajouter le paramètre β au filtrage, et de séparer en deux de la règle `dummy_iota_case`.

Le théorème 3.29 se généralise naturellement aux clôtures transitives et par contexte faible de `redn_term` et permet de prouver que la réduction des termes extraits termine toujours :

Lemme 3.31 (`Pe_t_wctxt_correct`)**.** *Soit t un terme* $\mathrm{CIC_B}$ *clos et bien typé. Si $t \to_{\mathcal{E}} t'$ et $t' \to^+_{\mathrm{B}'\mathcal{W}} u'$, alors il existe u tel que $u \to_{\mathcal{E}} u'$ et $t \to^+_{\mathrm{B}\mathcal{W}} u$.*

Preuve informelle. Par induction sur $t' \to^+_{\mathrm{B}'\mathcal{W}} u'$. □

Corollaire 3.32 (`Pe_sn_compat`)**.** *Soit t un terme* $\mathrm{CIC_B}$ *clos et bien typé. Si $t \to_{\mathcal{E}} t'$ et t normalise fortement pour $\to_{\mathrm{B}\mathcal{W}}$, alors t' normalise fortement pour $\to_{\mathrm{B}'\mathcal{W}}$.*

Preuve informelle. Par induction sur la preuve d'accessibilité de t. □

En combinant ce dernier résultat avec l'axiome 3.20, on peut en déduire que tous les programmes issus de l'extraction de termes bien typés terminent.

3.2.4 Fonction d'extraction

La définition de l'extraction par relation n'est pas directement exécutable. En effet, tous les résultats précédents supposent un terme extrait donné. Pour obtenir une fonction d'extraction concrète, on peut montrer que tout terme bien typé est extractible :

Théorème 3.33 (`extract_term`)**.** *Soit t un terme* $\mathrm{CIC_B}$ *bien typé dans un environnement Γ. Il existe t' tel que $t \to_{\mathcal{E}} t'$.*

Preuve informelle. Soit T le type principal de t. Si $\Gamma \vdash T : \mathsf{Prop}$, alors on choisit $t' = \square$ et on utilise `Pe_Prop`. Sinon, on choisit la règle de `Pe` dont le côté de la règle gauche correspond à t et on procède récusivement sur les sous-termes de t. □

On aurait aussi pu ne jamais utiliser `Pe_Prop` dans cette preuve. Mais cette preuve d'existence peut être vue comme une fonction d'extraction de $\mathrm{CIC_B}$ vers $\mathrm{CIC_{B'}}$ compatible avec la relation étudiée précédemment. On a alors intérêt à ce que le terme extrait soit le plus fin possible. En fait, le type exact de `extract_term` en Coq est :

```
forall sg, wf_sign sg -> forall e t,
  (exists T, sg -- e |- t : T) -> { t' | Pe e t t' }
```

Il est tel que le terme t' obtenu est informatif, et son extrait informel est une fonction (en OCaml) prenant en argument un terme de $\mathrm{CIC_B}$ et produisant un terme de $\mathrm{CIC_{B'}}$. En notant $\mathcal{E}$ cette fonction réifiée, on a ainsi :

$$t \to_{\mathcal{E}} \mathcal{E}(t)$$

3.3 Applications

On peut appliquer l'extraction informelle à toute la formalisation de $\mathrm{CIC_B}$ décrite dans ce chapitre, et en particulier à l'extraction formelle. Nous avons déjà mentionné le système autonome minimaliste de B. Barras ; nous y avons ajouté l'extraction (section 3.3.1). Indépendamment de la formalisation de l'extraction, nous proposons (section 3.3.2) d'utiliser du code extrait directement dans un greffon.

3.3.1 Système autonome minimaliste

Nous avons intégré la fonction d'extraction extraite à l'assistant de preuve minimaliste de B. Barras (Bcoq). À titre d'illustration, nous allons commenter une session de l'interpréteur mettant en œuvre cette extraction formelle.

L'invite de l'interpréteur est `Bcoq <`, et chaque commande entrée par l'utilisateur se termine par un point. Nous commençons par déclarer le type des entiers unaires :

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

Puis nous définissons l'addition sur les entiers (voir la section 2.1.3 pour la version Coq) :

```
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.
```

La syntaxe des termes est différente de celle de Coq et date de sa version 6 (nous n'avons pas touché à l'analyseur syntaxique de B. Barras) : « `fun (x:A) =>` » se notait alors « `[x:A]` », et « `forall (x:A),` » se notait alors « `(x:A)` ». Le « `Fix {X, F | ... }` » construit un paquet de points fixes à un élément. Dans le corps du point fixe, la variable de marque est nommée `X` et la variable pour les appels récursifs est nommée `F`. L'argument récursif n'a pas de dépendances, ce qui se traduit, dans CIC_B, par un paramètre de type `Lmark` (liste vide) qui n'apparaît pas dans la syntaxe concrète choisie par B. Barras. Le type du point fixe vu de l'intérieur (utilisé pour les appels récursifs) est `Lmark->(nat-X)->nat->nat`, celui vu de l'extérieur est `Lmark->nat->nat->nat` et le corps prend, outre le paramètre de type `Lmark`, un argument `n` de type `nat+X`. Le prédicat de retour du filtrage est indiqué entre `< ... >`. Dans la branche `S`, le prédécesseur de `n` est de type `nat-X`. Pour pouvoir véritablement utiliser la fonction d'addition, il est nécessaire de réaliser une projection (l'opérateur `Fix` construit une liste à un élément), et d'appliquer le résultat à un paramètre de type `Lmark`, ce qui explique le `F^0 ~` dans le corps et la construction similaire présente autour. Le type final de `plus` est bien celui attendu :

```
Bcoq < Check plus.
Type infere: nat->nat->nat
```

Pour illustrer l'extraction, nous avons ajouté une commande sommaire d'affichage :

```
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,~)),~)))),~)) ~)
```

Cet affichage n'est pas idéal, mais suffisant pour les besoins de notre démonstration. On remarquera que les notations `nat+X` et `nat-X` sont dépliées. Enfin, nous avons ajouté la commande d'extraction :

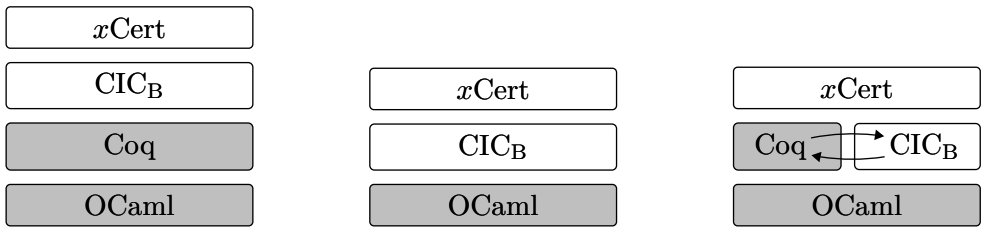
```
Bcoq < Extraction plus.
Extraction de plus:
(P1 Fix(#,[X:#]
  [F:#]
  ([_:#][n:#]Case{O|S}(n,(#,
    ([_0:#][m:#]m,
      ([pn:#][m:#]
        Cstr{S}(#,(P1 F # P1 pn m,#)),#)))),#)) #)
```

Dans le cas de `plus`, l'extraction consiste essentiellement à effacer les types.

3.3.2 Incorporation directe au sein de Coq

La fonction `extract_term` et, de manière plus générale, toutes les fonctions formalisées en Coq pourraient être exécutées directement au sein de Coq [2] en utilisant le noyau informel. Dans ce cas, un hypothétique xCert certifié utilisant CIC_B reposerait sur deux couches (logicielles) de confiance : le noyau de Coq, et la couche OCaml (incluant compilateur et environnement d'exécution). En utilisant Bcoq, on peut enlever la couche Coq, mais on perd toutes les facilités de Coq (syntaxe, notations, tactiques, extensibilité...), à moins de dupliquer (et d'adapter) toute l'infrastructure existante, ce qui pose des problèmes de maintenance à long terme.

Nous proposons d'inclure directement le noyau extrait à un greffon. Nous avons développé un prototype pour concrétiser cette idée, que nous avons appelé Xcoq. Le système embarque alors deux noyaux théoriquement indépendants : le noyau informel, non certifié, et le noyau extrait, certifié.



CIC_B dans Coq | Bcoq | Xcoq

Remarquons que quelle que soit l'approche, on repose sur la cohérence logique du Coq existant, tant que CIC_B n'aura pas été certifié avec lui-même (et des axiomes).

Avec l'approche Xcoq, le système existant peut ainsi servir d'interface au noyau extrait. Pour illustrer cela, rappelons la déclaration de `True` en Coq :

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

En syntaxe concrète Coq, la déclaration CIC_B correspondante s'écrit :

```
Definition dTrue := MutInductive (
  {|
    i_name := mk_str "True";
    i_param := mk_cterm _ Lmark;
    i_arg := mk_cterm _ Lmark;
    i_ckind := so;
    i_kind := so;
```

2. Toutefois, l'intrication des fonctions informatives et des définitions logiques opaques rend les calculs impossibles en pratique...

```
    i_constrs := Some (
      {|
        c_name := mk_str "I";
        c_arg := mk_cterm _ Lmark;
        c_inst := mk_cterm _ Nil_mark
      |} :: nil
    )
  |} :: nil
).
```

Ici, on déclare une constante `dTrue` en utilisant le noyau informel, constante qui *représente* la déclaration d'un inductif nommé `True` pour le noyau extrait. À ce stade, nous n'avons pas utilisé de greffon ; on pourrait par exemple appeler la fonction d'extension de signature (appelée `enter_decl` dans le développement) de $\mathrm{CIC_B}$ avec une signature vide et `dTrue` pour obtenir une signature non vide (*i.e.* un terme Coq de type `sigma`) prête à être utilisée pour typer des termes mettant en jeu `True` et `I`. À l'instar du noyau intégré à Coq, notre greffon maintient en OCaml une signature (*i.e.* une valeur OCaml qui pourrait correspondre à l'extrait informel d'un terme Coq de type `sigma`) et fournit une commande vernaculaire `XDeclare` permettant d'étendre cette signature en faisant appel à la version extraite d'`enter_decl`, et une commande `XCheck`, sosie de la commande `Check` intégrée. Par exemple, après exécution de :

```
XDeclare dTrue.
```

le type inductif `True` est enregistré auprès du noyau extrait et peut être référencé ultérieurement :

```
Coq < Definition xTrue :=
Coq <   Ind {mk_str "True" // Nil_mark,, Nil_mark,, Nil_mark}.
xTrue is defined
Coq < XCheck xTrue.
Ind  {{| string_value := "True" |} // Nil_mark,, Nil_mark,, Nil_mark}
     : %%(Sprop Neg)

Coq < Definition xI :=
Coq <   Cstr {mk_str "I" // Nil_mark,, Nil_mark}.
xI is defined
Coq < XCheck xI.
Cstr  {{| string_value := "I" |} // Nil_mark,, Nil_mark}
     : Ind  {{| string_value := "True" |} //
             Nil_mark,, Nil_mark,, Nil_mark}
```

Avec cette approche, on « transmet » des données du noyau informel vers le noyau formel, mais il serait également envisageable de faire l'inverse : une commande vernaculaire `XExtraction` pourrait par exemple faire appel à l'extraction extraite, et déclarer auprès du noyau informel une constante correspondant au résultat de l'extraction. Une autre commande pourrait également générer un fichier inspectable correspondant à la signature courante. Bcoq pourrait ainsi jouer le même rôle que le vérificateur autonome déjà présent dans Coq.

3.4 Un polymorphisme dangereux

Dans $\mathrm{CIC_B}$ comme dans Coq, Prop est un sous-type de Set et de Type_n. Cela veut dire qu'un terme peut être logique dans un contexte et informatif dans un autre, le passage de l'un à l'autre se faisant silencieusement. Par exemple, dans le terme suivant :

```
(fun A : Type => ...) True
```

la fonction s'attend à recevoir un terme informatif alors que `True` est logique. Cette promotion silencieuse de Prop en Type_n ne pose pas de problème dans le cadre de l'étude réalisée à la section précédente. Cependant, notre extraction laisse un résidu ($\Box$) pour chaque sous-terme logique effacé et l'extraction informelle applique des optimisations (décrites à la section 4.3 de [33]) sur les termes extraits en effaçant certains résidus. Nos tentatives d'optimisation se sont heurtées à la forme de polymorphisme qu'offre la promotion silencieuse de Prop en Type_n.

Par exemple, on aimerait pouvoir supprimer les champs logiques d'un enregistrement (ce qui revient, en Coq, à supprimer les arguments logiques des constructeurs d'inductifs), mais cela pose problème avec les enregistrements qui contiennent un champ dont le type est une grande sorte, comme dans :

$$\lambda A : \mathsf{Type}_n.\lambda X : (\!|\{x, y\} : \{A, \mathtt{nat}\}|\!).\mathsf{Constr}\{\mathsf{S} \mid \varepsilon, \mathtt{Field}(y)(X) :: \varepsilon\}$$

Dans la fonction ci-dessus, si A est instancié par `True`, l'argument X peut se retrouver excessivement simplifié au site d'appel, et la projection peut alors devenir incorrecte si on adopte un encodage par position. Ce contre-exemple est spécifique à la représentation de $\mathrm{CIC_B}$ et ne s'applique pas à Coq, mais montre bien l'importance de détails de représentation qui n'étaient pas pertinents jusqu'à présent.

Cette réflexion nous a mené à observer que le polymorphisme de sorte de Coq (inexistant dans $\mathrm{CIC_B}$) posait un problème vis-à-vis de l'extraction. En effet, l'extraction finalement implantée dans Coq omet complètement le filtrage sur inductif logique (c'est une des optimisations mentionnées ci-dessus). P. Letouzey justifie cette omission par le fait que les sortes mises en jeu lors d'un filtrage peuvent être déterminées statiquement grâce aux déclarations d'inductifs. C'est effectivement le cas dans le cadre de $\mathrm{CIC_B}$,

ainsi que dans le CIC sans polymorphisme de sorte considéré par P. Letouzey dans [33], ce qui correspond à Coq 8.0 et antérieur. Par contre, cela devient faux en présence du polymorphisme de sorte, et plus précisément du cas particulier suivant : un inductif à un seul constructeur peut avoir des instances particulières dans la sorte `Prop` même si la définition de l'inductif est *a priori* dans `Type`. Il suffit pour cela que cette instance du constructeur correspondant n'ait que des paramètres dans `Prop`, le système exploitant alors le cas particulier de l'inductif singleton. Considérons l'exemple suivant :

```
Definition G (A : Type) (x : A) (X : (A -> A) * True) :=
  (fst X x, 0).
Definition boum :=
  G True I ((fun x => x), I).
```

Les opérateurs infixes `,` et `*` représentent le constructeur (`pair`) et le type (`prod`) de la conjonction dans `Type`, et `fst` est la première projection. Ici, cette conjonction dans `Type` reçoit deux arguments logiques. Avec Coq entre 8.1 et 8.3pl2, l'extraction récursive de `boum` en OCaml donne :

```
let __ = let rec f _ = Obj.repr f in Obj.repr f

let g x x0 =
  Pair ((fst x0 x), 0)

let boum =
  g __ (Obj.magic __)
```

La valeur OCaml `__` est la concrétisation de □ choisie par P. Letouzey. Il s'agit d'une fonction ne faisant rien d'autre que d'accepter (et d'ignorer) des arguments. En particulier, il n'est (en théorie) pas possible d'en faire la projection. La projection ne produit pas d'erreur, mais il suffit d'appliquer le résultat (`fst x0 x`) pour obtenir une erreur de segmentation. La version Haskell plante plus gracieusement : la concrétisation de □ est une exception qui est lancée dès la projection (`fst x0`).

P. Letouzey a proposé [3] un contournement du problème en détectant l'instanciation à `Prop` des paramètres d'inductifs polymorphes singletons lors de l'extraction :

```
Coq < Recursive Extraction boum.
Error: The informative inductive type prod has a Prop instance.
This happens when a sort-polymorphic singleton inductive type
has logical parameters, such as (I,I) : (True * True) : Prop.
The Ocaml extraction cannot handle this situation yet.
Instead, use a sort-monomorphic type such as (True /\ True)
or extract to Haskell.
```

3. révision 14256 du dépôt Subversion de Coq

Ainsi, l'extraction échoue plutôt que de générer un programme gravement erroné. Ce correctif est disponible depuis la version 8.4, et a également été appliqué dans la version 8.3pl3.

Notons que les paires primitives de CIC_B sont équivalentes au `conj` polymorphe de Coq. Le `boum` ci-dessus, transposé à CIC_B, montre la nécessité des règles `dummy_proj1` et `dummy_beta` dans $\text{CIC}_{\text{B}'}$.

En réalité, un développement donné possède toujours un nombre fini de sortes (instances de `Type`), et il ne semble pas déraisonnable de dupliquer les inductifs et constantes polymorphes pour toutes les instances de `Type` possibles, afin de se ramener à des sortes ordonnées explicitement (le polymorphisme en de multiples paramètres de type `Type` est plutôt rare). En pratique, toutes les bibliothèques fournies en standard avec Coq ne nécessitent que quatre sortes `Type` différentes, et cela peut être rendu explicite avec la commande vernaculaire `Print Sorted Universes` que nous avons ajoutée à Coq et qui est disponible depuis la version 8.4. De plus, il semble suffisant de restreindre les instanciations de `Type` à `Set` et à `Prop`. Ainsi, la fonction `G` ci-dessus se déclinerait à l'extraction en deux versions, selon que `A` représente un type informatif ou logique.

Une autre solution serait de rendre explicites les promotions de `Prop` vers `Type` (et `Set`) dans la théorie, comme l'a fait P. Letouzey avec son CCI_m. La section 1.2.5 de [33] montre que cet ajout n'est pas trivial, et il n'est pas clair que la théorie ainsi obtenue soit équivalente à la précédente. Nous pensons néanmoins que ces promotions pourraient être inférées lors du pré-typage, ce qui permettrait de réduire l'impact du point de vue de l'utilisateur final. Notre connaissance limitée du pré-typage de Coq ne nous a pas permis d'implanter concrètement cette idée.

3.5 Bilan

3.5.1 Différences avec Coq

La principale différence, à la fois syntaxique et sémantique, entre CIC_B et ce qui est implanté dans Coq est la gestion de la terminaison. Coq utilise une condition de garde syntaxique plutôt évoluée et totalement détachée du typage, alors que CIC_B introduit des artefacts syntaxiques dans le typage. La relation entre ces deux procédés n'est pas immédiate, et une étude plus détaillée dépasserait le cadre de cette thèse. Cette différence ne semble pas avoir d'impact sur l'extraction.

CIC_B n'autorise pas les paramètres non récursivement uniformes. En particulier, le prédicat d'accessibilité `Acc` n'est pas directement disponible comme il l'est en Coq actuellement. Nous pensons toutefois qu'il serait possible d'autoriser à peu de frais ces paramètres non récursivement uniformes.

CIC_B ne dispose pas d'inductifs emboîtés, ni de types co-inductifs, mais ces fonctionnalités ne sont pas nécessaires à la formalisation de l'extraction.

3.5.2 Contributions

Nous avons formalisé en Coq l'extraction de $\mathrm{CIC_B}$ vers un langage abstrait, $\mathrm{CIC_{B'}}$, pouvant servir de base commune à plusieurs langages concrets. Nous avons prouvé en Coq la préservation de la sémantique lors de l'extraction : termes sources et termes extraits ont le même comportement.

Notre extraction est comparable à une phase de l'extraction implantée dans Coq. À notre connaissance, l'extraction de Coq n'a pas été l'objet d'étude théorique plus poussée depuis [33].

Nous avons également proposé une nouvelle application de l'extrait du développement, complémentaire à l'assistant de preuve minimaliste de B. Barras.

Bien que le langage source ne soit pas exactement le même, cette formalisation nous a permis de repérer une lacune dans la preuve originale de P. Letouzey : le cas où le redex est un filtrage sur inductif logique (à plusieurs constructeurs) produisant un terme logique a été omis. Ce cas peut se produire car l'effacement d'un sous-terme logique est facultatif avec la relation d'extraction. Notre solution a été de distinguer syntaxiquement les deux types de filtrages, et d'ajouter la règle `dummy_iota_case` à $\mathrm{CIC_{B'}}$ (seule une règle équivalente à `dummy_iota_case_pi` était présente dans le $\mathrm{CCI_\Box}$ de P. Letouzey).

Nous avons également remarqué le problème que posait le polymorphisme de sorte avec l'implantation actuelle de l'extraction. Bien que ce qui a été formalisé en Coq s'éloigne de ce qui est implanté dans Coq, le travail présenté ici nous a permis de trouver un problème significatif dans l'implantation, présent depuis plusieurs années.

Ce travail repose sur un développement de B. Barras présenté dans [5]. Nous avons procédé à beaucoup de réorganisation du code, ce qui nous a mené à modifier une bonne partie du développement. Le développement final fait environ 25 500 lignes de Coq. Le code extrait correspondant fait environ 5 000 lignes d'OCaml, auxquelles ont été ajoutées 1 000 lignes pour Bcoq et 900 lignes pour Xcoq.

3.5.3 Imprécisions et limitations

Le développement original contenait déjà des résultats admis, les plus importants étant la confluence et la décidabilité du sous-typage. Pour l'assistant de preuve minimaliste (Bcoq), la fonction de sous-typage était directement écrite en OCaml ; nous l'avons réécrite en Coq de sorte qu'elle s'extraie vers la version originale, mais toutes les parties logiques ont été admises.

Comme annoncé à la section 3.1.6, le lemme d'inversion 3.22 ne figure pas dans le développement Coq. Cependant, de multiples instances de ce lemme ont été admises au fil du développement. L'absence de ce lemme est due à la difficulté d'exprimer la condition sur les points fixes totalement appliqués. À des fins exploratoires, une branche alternative du développement forçant syntaxiquement les points fixes à être appliqués a été commencée, et a permis de former un énoncé et d'esquisser une preuve convaincante,

mais tout le développement n'a pas été adapté en conséquence.

La notion de clôture utilisée pour les termes est beaucoup trop restrictive : elle interdit notamment l'utilisation de constantes. Cette mesure simplificatrice a permis de se concentrer sur l'extraction d'un terme hors de tout contexte. Il manque une généralisation de l'extraction aux environnements et aux signatures.

Notre preuve de décidabilité de la condition d'application de la règle `Pe_Prop` n'est pas complète : l'algorithme implanté dans la fonction d'extraction regarde si le type principal d'un terme est de type `Prop` pour décider s'il doit l'effacer ou pas. Nous n'avons pas de preuve que dans le cas négatif, le terme n'est pas logique. Toutefois, cela ne nuit pas à la correction de la fonction d'extraction.

L'extraction de P. Letouzey efface aussi les schémas de type, mais nous n'avons pas du tout considéré cette règle.

3.5.4 Auto-génération

Une traduction automatique de Coq vers $\mathrm{CIC_B}$ permettrait théoriquement d'utiliser le développement Coq sur l'extraction pour générer l'extraction de Coq (procédé assimilable au *bootstrap*, ou auto-génération, d'un compilateur). Cependant, la différence entre les deux langages est trop significative pour réaliser une telle traduction.

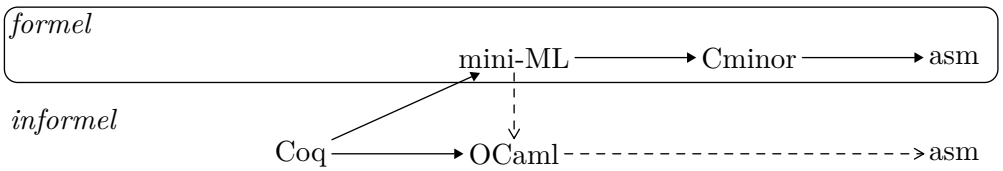
Il faut également mentionner que le langage cible étudié ici, $\mathrm{CIC_{B'}}$, *n'est pas OCaml*, ni un langage actuellement disponible comme cible de l'extraction informelle. En particulier, aucune stratégie de réduction n'est spécifiée et $\Box$, ainsi que les règles `dummy_` associées, n'ont aucune contrepartie directe dans les langages de programmation visés. L'implantation de $\Box$ peut se régler naïvement en enrobant toutes les valeurs issues de l'extraction dans une coquille indiquant leur type (fonction, type inductif ou objet logique effacé), et en effectuant des tests dynamiques lors de l'application de fonction et du filtrage. Une étude plus approfondie est nécessaire pour une implantation efficace, ainsi que pour juger de l'impact du choix d'une stratégie de réduction particulière.

4 Extraction interne

Dans le chapitre précédent, nous avons prouvé une correction universelle : *tous* les termes extraits sont corrects, quels que soient les termes originaux. Notre étude a porté sur le cœur du processus d'extraction : l'effacement des parties logiques. Nos preuves formelles ont été réalisées avec une modélisation de CIC différente de ce qui est implanté dans Coq, et nous avons arrêté notre étude à un langage cible assez proche du langage source. La mise en œuvre d'une passerelle complète entre le monde formel — entièrement défini dans Coq — et le monde informel — implanté concrètement et exécutable par un ordinateur — est délicate et demanderait encore plus de travail.

Dans ce chapitre, nous adoptons une approche plus pragmatique en proposant une extraction directe à partir de Coq. La correction des programmes extraits sera établie au cas par cas, et exprimée au sein même de l'assistant de preuve. Cette approche, que nous appelons *extraction interne*, est comparable au *proof-carrying code* [35] : chaque terme extrait est accompagné d'une preuve reliant sa sémantique au terme d'origine. Idéalement, cette preuve sera générée automatiquement. Ainsi, les propriétés vérifiées par le terme d'origine pourront être transposées au terme extrait sans que le processus d'extraction ne soit lui-même certifié.

Notre langage cible est une variante non typée de ML que nous appellerons *mini-ML*. Il s'agit, en première approximation, du langage source εML de [17], pour lequel Z. Dargaye a réalisé un compilateur vers Cminor, écrit et prouvé en Coq. Cminor est un langage intermédiaire de CompCert [31], un compilateur de C certifié ; ainsi, il est possible de compiler du mini-ML vers de l'assembleur directement exécutable par un ordinateur tout en restant dans le monde formel. Mini-ML correspond à un sous-ensemble d'OCaml, et il serait également envisageable de passer par une chaîne de compilation plus traditionnelle sur les plates-formes non entièrement supportées par un compilateur certifié.



Suivant le modèle du chapitre précédent, nous avons ajouté à notre mini-ML une constante □ destinée à remplacer les sous-termes logiques dans les programmes extraits. Toutefois, la sémantique opérationnelle n'a pas été adaptée en conséquence, et le travail présenté ici se concentre sur les programmes ne nécessitant pas les règles de réduction supplémentaires liées à □.

Nous présentons dans un premier temps le langage cible (section 4.1), puis nous détaillons un peu plus le principe de l'extraction interne sur un exemple (section 4.2). Nous proposons ensuite (section 4.3) une notion de correction sur les programmes extraits permettant de les relier à leurs termes d'origine, puis nous exposons (section 4.4) notre démarche pour prouver cette correction. Ce chapitre se termine par un bilan (section 4.5) des travaux réalisés.

Le travail exposé ici s'inspire faiblement de l'étude sémantique de l'extraction de P. Letouzey (section 2.4 de [33]). Cependant, plutôt que de nous placer dans un cadre idéalisé et complet, nous travaillons directement au sein même du système Coq, avec une classe restreinte de programmes, incluant les programmes qui auraient pu être directement écrit dans un ML typé. Bien que le travail sur l'extraction interne ait été initié en master [22], la démarche pour relier les termes extraits aux termes d'origine présentée ici est une contribution originale de cette thèse.

4.1 Langage cible

Le langage cible de l'extraction interne est une variante de ML, appelée ici *mini-ML*, proche du εML avec μ de Z. Dargaye [17].

Définition 4.1 (type `term`)**.** Les *termes* mini-ML sont définis par la grammaire suivante :

$$\begin{array}{rcl} t_1, t_2 : \texttt{term} & ::= & \natural n \\ & & \mid \mathsf{let}\ t_1\ \mathsf{in}\ t_2 \\ & & \mid \mathsf{fun}\ t \mid \mathsf{fix}\ t \mid t_1 t_2 \\ & & \mid \mathsf{constr}(n, \vec{t}) \mid \mathsf{match}(t, \vec{p}) \\ & & \mid \square \\ p : \texttt{pat} & ::= & (n, t) \end{array}$$

où $n \in \mathbb{N}$.

On y retrouve toutes les constructions calculatoires de CIC :

- les variables, représentées par des indices de de Bruijn ;
- les définitions locales. Dans $\mathsf{let}\ t_1\ \mathsf{in}\ t_2$, une nouvelle variable est liée à t_1 dans t_2 ;
- les fonctions. Dans $\mathsf{fun}\ t$, une nouvelle variable référençant l'argument est liée dans t ;
- les points fixes. Dans $\mathsf{fix}\ t$, deux nouvelles variables sont liées dans t : une pour le premier argument et une pour le point fixe lui-même ;
- l'application de fonction ;
- les constructeurs. On suppose que tous les constructeurs d'un type inductif sont numérotés séquentiellement et qu'un constructeur est toujours complètement appliqué. Par exemple, l'entier unaire 1 est noté :

$$\mathsf{constr}(1, \mathsf{constr}(0))$$

– le filtrage. Dans $\mathsf{match}(t, \vec{p})$, t est le terme filtré, et chaque motif de $\vec{p}$ est composé de l'arité du constructeur associé (la liste est indexée par numéro de constructeur) et d'un corps (l'arité indique le nombre de variables liées dans le corps).

On retrouve de plus le terme spécial $\square$, servant à représenter les termes logiques effacés, comme au chapitre 3. Concrètement, `term` est défini en Coq comme suit :

```
Inductive term : Set :=
  | TDummy : term
  | TVar : nat -> term
  | TLet : term -> term -> term
  | TFun : term -> term
  | TFix : term -> term
  | TApply : term -> term -> term
  | TConstr : nat -> list term -> term
  | TMatch : term -> list pat -> term
with pat : Set :=
  | Patc : nat -> term -> pat.
```

Mini-ML se veut être un langage proche des différentes variantes de ML implantées concrètement. La correction des programmes extraits utilise une relation de réduction. Cette dernière fait appel à une notion de valeur :

Définition 4.2 (prédicats `IsValue`, `IsValue_list`)**.** Les *valeurs* de mini-ML sont les suivantes :

$$\frac{}{\mathsf{val}\ \square} \qquad \frac{}{\mathsf{val}\ (\mathsf{fun}\ t)} \qquad \frac{}{\mathsf{val}\ (\mathsf{fix}\ t)}$$

$$\frac{\mathsf{val}\ \vec{t}}{\mathsf{val}\ \mathsf{constr}(n, \vec{t})} \qquad \frac{}{\mathsf{val}\ []} \qquad \frac{\mathsf{val}\ t \quad \mathsf{val}\ \vec{u}}{\mathsf{val}\ (t :: \vec{u})}$$

Voici la relation de réduction proprement dite, définie simultanément sur les termes et les listes de termes :

Définition 4.3 (relations `SmallStep`, `SmallStep_list`)**.** La *réduction* de mini-ML est l'union des règles suivantes :

$$\frac{\mathsf{val}\ u}{\mathsf{let}\ u\ \mathsf{in}\ t \Rightarrow t\,\{\natural 0 \leftarrow u\}}$$

$$\frac{\mathsf{val}\ u}{(\mathsf{fun}\ t)u \Rightarrow t\,\{\natural 0 \leftarrow u\}} \qquad \frac{\mathsf{val}\ u}{(\mathsf{fix}\ t)u \Rightarrow t\,\{\natural 0 \leftarrow u\}\,\{\natural 0 \leftarrow \mathsf{fix}\ t\}}$$

$$\frac{\mathsf{val}\ \vec{t} \qquad p_n = (k, u) \qquad k = \left|\vec{t}\right|}{\mathsf{match}(\mathsf{constr}(n, \vec{t}), \vec{p}) \Rightarrow u\,\{\natural 0 \leftarrow t_1\} \cdots \{\natural 0 \leftarrow t_k\}}$$

$$\frac{t \Rightarrow t'}{t\ u \Rightarrow t'\ u} \qquad \frac{u \Rightarrow u'}{t\ u \Rightarrow t\ u'} \qquad \frac{t \Rightarrow t'}{\mathsf{let}\ t\ \mathsf{in}\ u \Rightarrow \mathsf{let}\ t'\ \mathsf{in}\ u}$$

$$\frac{t \Rightarrow t'}{\mathsf{match}(t, \vec{p}) \Rightarrow \mathsf{match}(t', \vec{p})} \qquad \frac{\vec{t} \Rightarrow \vec{t'}}{\mathsf{constr}(n, \vec{t}) \Rightarrow \mathsf{constr}(n, \vec{t'})}$$

$$\frac{t \Rightarrow t'}{t, \vec{u} \Rightarrow t', \vec{u}} \qquad \frac{\vec{u} \Rightarrow \vec{u'}}{t, \vec{u} \Rightarrow t, \vec{u'}}$$

Ces règles munissent mini-ML d'une stratégie d'évaluation en appel par valeur. Les quatre premières règles correspondent aux différents redex possibles, les règles restantes sont des règles de passage au contexte.

`SmallStep` est une sémantique opérationnelle à petits pas avec substitutions. Une autre sémantique équivalente — opérationnelle à grands pas avec environnement — avait été initialement envisagée [22], mais nous l'avons jugée moins pratique dans le cadre de l'extraction interne.

L'équivalent des règles `dummy_` de $\mathrm{CIC}_{\mathrm{B}'}$ manquent : $\Box$ est une constante inerte. Ces règles peuvent être vraiment nécessaires dans certains cas, mais le travail présenté ici n'a pas atteint un stade où leur absence (ou leur présence) était significative, donc nous avons choisi de rester fidèle à εML.

4.2 Principe de l'extraction interne

Nous avons écrit (en OCaml) une fonction d'extraction qui efface les sous-termes logiques de la manière décrite dans la preuve du théorème 3.33, ainsi que les types. Elle se présente sous la forme d'un greffon qui crée une nouvelle commande vernaculaire, `Internal Extraction`. Il s'agit d'une version épurée de l'extraction existante, qui produit directement un terme Coq de type `term`, au lieu d'une chaîne de caractères. Il devient alors possible de raisonner sur les programmes extraits au sein même de l'assistant de preuves. Voici un exemple de retranscription d'une session interactive :

```
Coq < Print plus.
plus =
fix plus (n m : nat) : nat :=
  match n with
  | 0 => m
  | S p => S (plus p m)
  end
     : nat -> nat -> nat

Coq < Internal Extraction plus.
```

```
The constant plus__extr has been created by extraction.

Coq < Print plus__extr.
plus__extr =
TFix
  (TFun
     (TMatch (TVar 1)
        (Patc 0 (TVar 0)
         :: (Patc 1
               (TConstr 1
                  (TVar 3 @ TVar 0 @
                   TVar 1 :: nil))
             :: nil)%list)))
     : term
```

La constante `plus__extr` créée par l'extraction interne à partir de la constante `plus` peut aussi être écrite sous la forme :

$$\mathsf{fix}\ (\mathsf{fun}\ (\mathsf{match}(\natural 1, (0, \natural 0), (1, \mathsf{constr}(1, \natural 3\ \natural 0\ \natural 1)))))$$

Le but de l'extraction interne est de prouver en Coq des résultats sur la sémantique opérationnelle de `plus__extr` en utilisant `plus`. On peut aussi dire que, dans un certain sens, `plus__extr` est construit de sorte que `plus` en soit une sémantique dénotationnelle. Lorsque la relation est établie, nous dirons que `plus__extr` *simule* `plus`. Le problème de l'extraction interne se résume à trouver une relation de simulation adéquate qui puisse se prouver facilement (dans l'idéal, automatiquement) dans le cas d'un programme extrait et de son terme Coq d'origine. Nous n'avons pas atteint le but ultime d'une implantation complète (c'est-à-dire traitant tous les termes Coq) et automatisant les preuves des programmes extraits. Néanmoins, nous allons présenter dans les sections suivantes un cadre générique développé dans ce but, ainsi que quelques exemples.

Notre cadre de certification des programmes extraits repose sur le système de tactiques de Coq. Dans la section 4.3, nous allons détailler la notion de correction des termes extraits que nous avons adoptée. Nous avons défini un jeu de tactiques permettant de prouver cette correction, que nous allons présenter à la section 4.4.

4.3 Extractibilité et prédicats de simulation

Dans cette section, nous présentons la relation de simulation utilisée pour exprimer la correction des termes extraits. Ce qui est présenté ici s'inspire de la section 2.4.2 de [33]. La principale nouveauté est l'explicitation de la réduction dans le langage cible, passée sous silence par P. Letouzey. Notre champ d'action est aussi plus restreint, mais nous travaillons dans le cadre concret du système Coq existant.

Définition 4.4 (prédicat `InternallyExtractible`)**.** Un type A est *extractible* s'il existe une relation `ie_pred` de type $A \to$ `term` $\to$ `Prop` telle que pour tous x et x', si `ie_pred` x x', alors x' est une valeur close. Ceci est implanté en Coq par une classe de types :

```
Class InternallyExtractible A := {
  ie_pred : A -> term -> Prop;
  ie_value : forall x x', ie_pred x x' -> IsValue x';
  ie_closed : forall x x', ie_pred x x' -> clos_after 0 x'
}.
```

Dans la définition ci-dessus, la relation `ie_pred` doit de plus coïncider avec la relation d'extraction vue au chapitre 3 sur les termes inductifs clos du premier ordre (c'est-à-dire sans sous-termes fonctionnels), mais cette condition n'est pas exprimable en Coq et donc absente de la définition formelle. Nous voulons ainsi écarter la relation toujours fausse, qui vérifie les contraintes imposées.

En Coq, une classe de types est un enregistrement, et la définition ci-dessus est à comparer avec les s^+ de P. Letouzey. Une différence majeure est le passage du type A en tant que paramètre plutôt que champ de l'enregistrement. Cela permet de bénéficier du polymorphisme de sorte tout en évitant les incohérences d'univers : avec la présentation de P. Letouzey, il faut en fait autant de `mk_Type` que de `Type` utilisés dans le développement. De plus, le passage de A en paramètre permet d'utiliser le mécanisme de classe de types de Coq : les instances de `InternallyExtractible` sont souvent laissées implicites, et recherchées en utilisant le paramètre comme clé.

Définition 4.5 (prédicat `extracted`)**.** Soit A un type extractible, x un terme de type A, et x' et x'' deux termes de type `term`. On dit que le triplet (x, x', x'') est *représentatif* si :

1. $x' \Rightarrow^* x''$;
2. `ie_pred` x x'' ;
3. x' est clos.

Ceci est implanté en Coq par un enregistrement :

```
Record extracted `{IEA : InternallyExtractible A} x x' x'' := {
  ie_ered : x' ===> x'';
  ie_epred : ie_pred x x'';
  ie_eclosed : clos_after 0 x'
}.
```

Dans la définition ci-dessus, $\Rightarrow^*$ et `===>` désignent la clôture réflexive et symétrique de `SmallStep`. L'annotation `` `{IEA : InternallyExtractible A} `` déclare deux paramètres implicites, `A` et `IEA`. Ainsi, concrètement, `extracted` n'a l'air de n'avoir que trois arguments, `x`, `x'` et `x''`.

Définition 4.6 (relation `simul`)**.** Soit A un type extractible, x un terme de type A, et x' un terme de type `term`. On dit que x' *simule* x s'il existe x'' tel que (x, x', x'') soit représentatif. Ceci est une simple définition en Coq :

```
Definition simul `{IEA : InternallyExtractible A} x x' :=
  exists x'', extracted x x' x''.
```

Les prédicats d'extractibilité seront définis au cas par cas selon le type. Les définitions `InternallyExtractible`, `extracted` et `simul` sont des enrobages qui ont pour but de faciliter les preuves de simulation. Ainsi définie, la relation de simulation `simul` est stable par anti-réduction et est naturellement conséquence de `ie_pred` :

Lemme 4.7 (`ie_from`)**.** *Soient x, x' et x'' tels que* `simul` $x\ x'$, $x'' \Rightarrow^* x'$ *et x'' clos. On a* `simul` $x\ x''$.

Lemme 4.8 (`ie_compat`)**.** *Soient x, x' tels que* `ie_pred` $x\ x'$. *On a* `simul` $x\ x'$.

De plus, on a la garantie que le terme mini-ML associé normalise. Elle va correspondre au $[\![\cdot]\!]_{\mathbb{1}}$ de P. Letouzey ([33], section 2.4.3). Notons que la réciproque de `ie_compat` est en général fausse.

Nous n'avons pas implanté de procédure complète pour prouver que tous les types sont extractibles. Nous allons présenter dans ce qui suit l'extractibilité de quelques exemples significatifs.

4.3.1 Types primitifs

Termes effacés Si un terme est logique ou un type, il est effacé par l'extraction, et sa valeur exacte est non pertinente. La relation d'extractibilité est alors très simple :

```
Definition is_TDummy_pred x x' := x' = TDummy.
```

Fonctions Soit A un type extractible, et B un type dépendant de A tel que pour tout x, $B\ x$ soit extractible. Alors `forall`$(x : A), B\ x$ est extractible. Le prédicat d'extractibilité est implanté par un enregistrement comme suit :

```
Record prod_pred (f : forall x, B x) f' := {
  ie_prod_value : IsValue f';
  ie_prod_clos : clos_after 0 f';
  ie_prod_pred : forall x x', ie_pred x x' -> simul (f x) (f' @ x')
}.
```

Cette définition est telle que prouver que f' simule f peut se ramener directement à prouver que $f'\ x'$ simule $f\ x$ pour toute *valeur* (close) x' qui simule x. De plus, pour

tout *terme* (clos, mais pas forcément une valeur) x' simulant x, $f'\ x'$ simule $f\ x$. Cela permet de prouver directement la simulation entre deux termes de forme applicative.

Au contraire de la relation d'extraction vue au chapitre 3, la relation de simulation ne s'intéresse aux fonctions que d'un point de vue extensionnel. La commande `Internal Extraction` fournit un programme extrait fidèle à la syntaxe du terme original, et les méthodes données ici pour prouver la simulation ne s'appliquent qu'aux extraits syntaxiques, mais il serait également possible de changer le programme extrait soit pour la version extraite d'une autre fonction Coq extensionnellement égale, soit directement par un terme mini-ML pouvant ne pas être l'image d'un terme Coq par l'extraction.

L'hypothèse d'extractibilité du type du domaine A n'est pas anodine : la solution proposée ici ne gère pas le cas où A possède des variables libres. En pratique, cette situation peut se présenter avec les types polymorphes et les types dépendants. Par exemple, l'application de fonction a le type suivant :

```
apply :
  forall (A : Type) (B : A -> Type),
    (forall x : A, B x) -> forall x : A, B x
```

Dans ces situations, on peut se ramener au cadre étudié en considérant la fonction partiellement appliquée à ses arguments qui apparaissent dans des types. Dans le cas d'`apply`, le lemme de simulation peut prendre la forme suivante :

```
Lemma apply__simul :
  forall A (IEA: InternallyExtractible A),
    forall B (IEB: forall x, InternallyExtractible (B x)),
      simul (@apply A B) (apply__extr @ TDummy @ TDummy).
```

et est prouvable en utilisant les techniques exposées dans la section 4.4. Nous pouvons comme cela traiter tous les cas de polymorphisme à la ML, où les types polymorphes sont sous forme prénexe. En particulier, les types dépendants comme `vect` ne sont pas totalement gérés.

4.3.2 Types inductifs

Entiers unaires Rappelons la définition des entiers unaires :

```
Inductive nat : Set :=  O : nat | S : nat -> nat.
```

Sur ce type de données du premier ordre, le prédicat d'extractibilité est une spécialisation de la relation d'extraction :

```
Inductive nat_pred : nat -> term -> Prop :=
  | nat_predO : nat_pred O (TConstr 0 nil)
```

```
  | nat_predS :
      forall n n', nat_pred n n' ->
        nat_pred (S n) (TConstr 1 (n'::nil)).
```

Sous cette forme, `nat_pred` ressemble beaucoup aux $\overline{\texttt{nat}}$ et $\widehat{\texttt{nat}}$ de P. Letouzey.

Remarquons que la réduction de mini-ML n'apparaît pas dans la déclaration de `nat_pred`, mais elle apparaît explicitement dans le prédicat final `simul`. Avec ces définitions, on peut prouver que $\mathsf{constr}(0)$ simule bien `O` :

```
Lemma O__simul : simul O (TConstr 0 nil).
```

et que $\mathsf{constr}(1, \cdot)$ simule bien $\texttt{S} \cdot$:

```
Lemma S__simul :
  forall n n', simul n n' ->
    simul (S n) (TConstr 1 (n'::nil)).
```

Ces deux énoncés correspondent aux constructeurs du $\overline{\texttt{nat}}$ de P. Letouzey.

Nous avons ainsi mis en place toute l'infrastructure pour énoncer la simulation de l'addition :

```
Lemma plus__simul : simul plus plus__extr.
```

Nous expliquerons à la section 4.4 comment prouver ce lemme.

Listes polymorphes Voici la définition des listes polymorphes en Coq :

```
Inductive list (A : Type) : Type :=
  nil : list A | cons : A -> list A -> list A
```

Tout comme `nat`, il s'agit d'un type de données récursif du premier ordre. Il est extractible, dès lors que son paramètre l'est :

```
Variable A : Type.
Hypothesis IEA : InternallyExtractible A.
```

Le prédicat d'extractibilité est construit sur le même modèle que `nat` :

```
Inductive list_pred : list A -> term -> Prop :=
  | nil_pred : list_pred nil (TConstr 0 nil)
  | cons_pred :
    forall x x', ie_pred x x' ->
      forall xs xs', list_pred xs xs' ->
        list_pred (x::xs) (TConstr 1 (x'::xs'::nil)).
```

On remarquera l'utilisation du prédicat générique `ie_pred`, qui utilise implicitement la variable de section `IEA` grâce aux classes de types. Il est ainsi possible d'énoncer et de prouver dans notre formalisme la simulation pour la fonction (polymorphe) qui calcule la longueur d'une liste :

```
Lemma length__simul :
  forall A (IEA : InternallyExtractible A),
    simul (@length A) (length__extr @ TDummy).
```

ou encore la fonction d'itération d'une fonction sur une liste :

```
Lemma map__simul :
  forall A (IEA : InternallyExtractible A),
    forall B (IEB : InternallyExtractible B),
      simul (@map A B) (map__extr @ TDummy @ TDummy)
```

4.3.3 Types partiellement logiques

Nos techniques permettent de gérer certains types partiellement logiques non dépendants. Cependant, les types dépendants sont problématiques de manière générale, et beaucoup de types partiellement logiques sont dépendants, ce qui limite notre approche.

Types non dépendants Le type des entiers pairs peut par exemple être déclaré en tant que type inductif :

```
Inductive even : Type :=
  even_intro : forall n, (exists p, n = p + p) -> even.
```

Il est informatif, mais le deuxième argument de son constructeur est logique et effacé par l'extraction. D'ailleurs, l'extraction informelle — après optimisation — traduit ce type directement vers `nat`. Ce type est extractible :

```
Inductive even_pred : even -> term -> Prop :=
  even_pred_intro :
    forall n n', ie_pred n n' ->
      forall h h', ie_pred h h' ->
        even_pred (@even_intro n h) (TConstr 0 (n'::h'::nil)).
```

Là encore, on exploite les classes de types en utilisant le prédicat générique `ie_pred` : une fois avec `nat`, une fois avec `(exists p, n = p + p)` (qui est purement logique). La fonction de doublement peut se définir à l'aide du système de tactiques :

```
Definition double : nat -> even.
Proof.
  refine (fun p => @even_intro (p + p) _).
  exists p; reflexivity.
Defined.
```

Le terme CIC correspondant est :

```
double =
  fun p : nat =>
    @even_intro (p + p) (ex_intro _ p eq_refl)
```

Ce qui donne après extraction :

```
double__extr =
  TFun
    (TConstr 0 (plus__extr @ TVar 0 @ TVar 0 :: TDummy :: nil))
```

On remarquera que le `(ex_intro _ p eq_refl)` — de type `(exists p0, p + p = p0 + p0)` — a été remplacé par `TDummy`. Il est alors possible d'énoncer et de prouver la simulation pour `double` :

```
Lemma double__simul : simul double double__extr.
```

Types dépendants Nous avons déjà évoqué dans le cas du produit dépendant le caractère problématique des types dépendants. Reprenons l'exemple du prédécesseur `pred` avec le type riche :

```
forall n, n <> 0 -> { p | n = S p }
```

Le type `{ p | n = S p }` est une instance de `sig`, qui est lui-même un type polymorphe et peut être dépendant. Pour simplifier, déclarons-en une version spécialisée :

```
Inductive predT n :=
  predT_intro : forall p, n = S p -> predT n.
```

`pred` peut alors être réécrit pour avoir le type :

```
forall n, n <> 0 -> predT n
```

Ce qui donne après extraction :

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

On peut bien définir une notion d'extractibilité pour `predT` :

```
Inductive predT_pred n : predT -> term -> Prop :=
  predT_pred_intro :
    forall p p', ie_pred p p' ->
      forall h h', ie_pred h h' ->
        predT_pred (@predT_intro _ p h) (TConstr 0 (p'::h'::nil)).
```

On peut également énoncer la simulation pour `pred` :

```
Lemma pred__simul : simul pred pred__extr.
```

Malheusement, ce lemme n'est pas prouvable automatiquement par les techniques décrites à la section 4.4 à cause du filtrage avec prédicat de retour complexe apparaissant dans `pred`.

4.3.4 Autres types

En Coq, tout terme typé par une sorte est un type, et il est par exemple possible de *calculer* des types, comme l'illustre le `nary` de la page 29. Cependant, notre étude ne porte que sur les types construits uniquement à partir de produits dépendants et de types inductifs, ce qui inclut notamment tous les types ML.

4.4 Preuves de simulation

En utilisant le mode preuve de Coq, nous pouvons prouver de façon incrémentale la relation de simulation entre un terme et sa version extraite, pour une classe restreinte de termes. Dans cette section, nous présentons des tactiques permettant d'évaluer symboliquement les termes Coq et les termes mini-ML en parallèle dans un but, tout en conservant la relation de simulation entre eux.

Ces tactiques permettent de construire des preuves dirigées par le but : la simulation entre deux termes composés est ramenée aux simulations entre chacune de leurs composantes, et cela, récursivement jusqu'aux termes atomiques. Cette décomposition peut faire apparaître des redex qui sont réduits grâce à la stabilité par anti-réduction.

Nous allons d'abord expliquer dans la section 4.4.1 le déroulement général des preuves de simulation, et en particulier notre gestion de l'environnement, puis nous allons détailler dans la section 4.4.2 notre approche réflexive pour réduire des termes mini-ML dans les conclusions de buts de simulation. Notre approche achoppe sur le filtrage, lui-même lié aux points fixes par la condition de garde ; nous donnons néanmoins quelques pistes dans la section 4.4.3.

4.4.1 Gestion de l'environnement

Les tactiques manipulent toujours des termes mini-ML clos au niveau objet (pas d'indice de de Bruijn orphelin). Pour cela, tous les termes ouverts apparaissant au cours d'une preuve sont immédiatement clos avec des variables au niveau meta. La forme générale d'un but est donc de la forme :

```
n : nat
n' : term
Hnn' : ie_pred n n'
m : nat
m' : term
Hmm' : ie_pred m m'
============================
 simul t t'
```

où `n` et `m` *sont* les variables libres de `t`, et `n'` et `m'` *représentent* les variables libres de `t'` et dénotent des termes clos abstraits. Nous qualifierons d'*hermétiques* les variables `n`, `n'` et `Hnn'`.

Dans le cas où `t` et `t'` sont des variables, on peut conclure directement par `ie_compat`. On a le même comportement pour les constantes vis-à-vis de l'environnement global : par exemple, si la définition de `mult` fait appel à celle de `plus`, la preuve de simulation reliant `mult` à `mult__extr` fera appel à celle reliant `plus` à `plus__extr`. La preuve de simulation des variables et des constantes peut se faire à l'aide de la tactique `auto` (qui effectue une recherche de preuve à la Prolog), combinée à une base d'indices contenant initialement `ie_compat`, et à laquelle sont ajoutés au fur et à mesure les résultats de simulation des constantes. Nous appelons `ie_easy` cette variante d'`auto`.

Dans le cas où `t` et `t'` sont des applications de fonction, une tactique `ie_apply` permet de se ramener à deux sous-buts : un concernant la fonction, et un concernant l'argument. Dans le cas où `t` et `t'` sont des fonctions, une tactique `ie_hnf` permet de rendre apparentes des variables hermétiques, qui peuvent ensuite être introduites avec `intros`. Les tactiques `ie_apply` et `ie_hnf` ne font qu'appliquer des transformations mineures au but.

4.4.2 Gestion de la réduction mini-ML par réflexion

Nous avons implanté des tactiques réflexives qui tirent parti de la forme particulière des buts rencontrés lors d'une preuve de simulation. Nous avons pour cela dupliqué la définition des termes en y ajoutant deux nouveaux constructeurs :

```
Inductive IE_term :=
  | IE_TDummy : IE_term
  | IE_TVar : nat -> IE_term
```

```
| IE_TLet : IE_term -> IE_term -> IE_term
| IE_TFun : IE_term -> IE_term
| IE_TFix : IE_term -> IE_term
| IE_TApply : IE_term -> IE_term -> IE_term
| IE_TConstr : nat -> list IE_term -> IE_term
| IE_TMatch : IE_term -> list (nat * IE_term) -> IE_term
| IE_Value :
    forall `{IEA : InternallyExtractible A} x x',
      ie_pred x x' -> IE_term
| IE_Extracted :
    forall `{IEA : InternallyExtractible A} x x' x'',
      extracted x x' x'' -> IE_term.
```

Nous avons implanté une *tactique* de réification `IE_quote` effectuant la traduction d'un `term` en un `IE_term`, et une *fonction* d'interprétation `IE_interp` traduisant un `IE_term` en un `term`.

`IE_quote` prend en paramètre un terme `t` et renvoie un terme de la forme `IE_interp u` convertible. `IE_quote` est trivial dans les cas où le terme traduit possède un constructeur apparent (ce dernier est transformé en sa version préfixée par `IE_`). Les autres cas correspondent aux variables hermétiques (ou à des constantes) pour lesquelles une hypothèse `ie_pred` existe, hypothèse qui est alors injectée dans la valeur retournée grâce au constructeur `IE_Value`. Des triplets représentatifs peuvent également apparaître lors des preuves, et sont capturées par `IE_quote` : `IE_Extracted x x' x'' H` représente le terme `x'`, mais encapsule aussi sa forme normale `x''`. La traduction inverse, `IE_interp`, est quant à elle directe.

Grâce au type `IE_term`, il devient possible d'écrire une fonction `IE_red` effectuant une réduction « symbolique » d'un terme mini-ML, c'est-à-dire tenant compte des variables hermétiques. Cette fonction prend en paramètre supplémentaire un entier naturel bornant le nombre d'appels récursifs. Nous avons prouvé le lemme suivant :

```
Lemma IE_red_correction :
  forall n u, IE_interp u ===> IE_interp (IE_red n u).
```

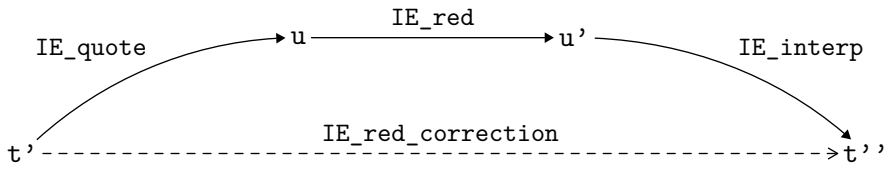
qui exprime le fait que la fonction d'évaluation `IE_red` définie sur les `IE_term` est bien compatible avec la relation de réduction `SmallSteps` (notée `===>`) définie sur les `term`.

En utilisant cette infrastructure, nous avons implanté une tactique `ie_simpl` dont la vocation est de réduire le terme mini-ML apparaissant dans la conclusion d'un but, de la même manière que `simpl` réduit les termes Coq. Plus précisément, face à un but de la forme `simul t t'`, cette tactique :

1. trouve un `u` tel que `IE_interp u` soit convertible à `t'` ;
2. évalue `IE_interp (IE_red 10 u)` pour obtenir un terme `t''` (10 est une valeur arbitraire, suffisamment grande en pratique) ;

3. applique le lemme `ie_from` afin de ramener le but à `simul t t''`. Les sous-buts annexes sont résolus grâce à `IE_red_correction` et d'autres lemmes similaires.

Le fonctionnement de `ie_simpl` peut être schématisé ainsi :



4.4.3 Filtrage et points fixes

Sur des exemples simples, la simulation entre un filtrage et sa version extraite peut se prouver avec un filtrage sur le prédicat d'extractibilité associé au terme filtré. Dans le cas du filtrage sur un entier `n` (dont la valeur extraite est `n'`, et l'hypothèse associée est `Hnn'`) avec un prédicat de retour trivial, il suffit d'abstraire la conclusion du but par rapport à `n` et `n'`, puis de raffiner le but avec un filtrage sur `Hnn'` ayant pour prédicat de retour la fonction apparaissant dans la conclusion. Par exemple, face au but :

```
n : nat
n' : term
Hnn' : ie_pred n n'
============================
 simul (match n with
 | O => n
 | S u => u
 end) (TMatch n' (Patc 0 n' :: Patc 1 (TVar 0) :: nil))
```

Les tactiques suivantes permettent l'analyse de cas :

```
pattern n at 1, n' at 1.
lazymatch goal with
  |- ?A _ _ =>
    refine (match Hnn' in nat_pred x x' return A x x' with
              | nat_predO => _
              | nat_predS a b Hie => _
            end)
end.
```

Deux sous-buts sont alors générés, correspondant aux deux cas de `nat`. On appelle `ie_destruct_nat` la deuxième tactique ; on peut définir de façon similaire une tactique `ie_destruct_list` et, plus généralement, une tactique `ie_destruct_`x pour tout type

inductif x n'ayant que des paramètres de type `Type` et pas d'indices, comme les types ML.

Nous avons utilisé la tactique `pattern` de Coq, qui permet de mettre un but sous la forme d'une application de fonction. Dans l'exemple ci-dessus,

```
pattern n at 1, n' at 1
```

abstrait le but par rapport aux *premières* occurrences de `n` et de `n'` pour le mettre sous la forme `A n n'`, comme le suggère le prédicat de retour de `ie_destruct_nat`. Dans le cas présent, la séquence des deux tactiques `pattern` et `ie_destruct_nat` est équivalente à un simple `destruct Hnn'`, qui effectue une analyse de cas sur `Hnn'`. Toutefois, l'utilisation directe de `destruct` ne convient pas en général car elle agit sur *toutes* les occurrences de `n` et `n'`, ce qui peut être problématique, par exemple avec la fonction suivante :

```
Fixpoint test1 n m {struct n} :=
  match n with
    | O => O
    | S p => test1 p (match p with
                        | O => O
                        | S q => test1 p q
                      end)
  end.
```

En effet, l'action de `destruct` sur l'occurrence de `p` figurant dans le filtrage de `p` casse la condition de garde pour l'appel récursif `test1 p q`.

En suivant cette idée, il serait possible d'écrire (en OCaml) une hypothétique tactique `ie_destruct` qui combinerait l'appel à `pattern` et à `ie_destruct_`x. Cette tactique pourrait être accessible à l'utilisateur via un greffon, et serait conceptuellement un sosie du `destruct` de Coq, mais n'agissant que sur l'occurrence d'une variable apparaissant en tête de filtrage.

Notre gestion du filtrage a été choisie pour préserver la condition de garde des points fixes : nos essais montrent que la simulation d'un point fixe dont l'argument décroissant est `n` peut être prouvée avec un point fixe dont l'argument décroissant est l'hypothèse reliant `n` à son extrait.

Enfin, `ie_destruct` ne permet pas de gérer les filtrages avec un prédicat de retour élaboré comme ceux que l'on peut rencontrer avec les types dépendants. Les exemples non triviaux sont déjà très gros, et rendent difficiles les expérimentations. En outre, notre approche ne permet de gérer que les points fixes commençant par un filtrage sur leur argument décroissant, après d'éventuelles abstractions. En particulier, les points fixes n'utilisant pas leur argument décroissant, ou ceux ayant un argument décroissant logique sont exclus d'un hypothétique traitement automatique.

4.4.4 Un exemple de preuve de simulation : l'addition

Nous allons illustrer toutes nos tactiques sur un exemple : la preuve de simulation pour l'addition. Rappelons tout d'abord sa définition :

Version Coq : `plus`	*Version mini-ML* : `plus__extr`

`(fix plus (n m : nat) : nat :=`
`  match n with`
`  | 0 => m`
`  | S p => S (plus p m)`
`  end)`
	`(TFix (TFun (TMatch (TVar 1)`
	`  (Patc 0 (TVar 0)`
	`  :: Patc 1 (TConstr 1`
	`  (TVar 3 @ TVar 0 @ TVar 1`
	`  :: nil))`
	`  :: nil))))`

Compte-tenu de toute l'infrastructure mise en place, voici l'énoncé et la preuve complète de simulation :

```
Lemma plus__simul : simul plus plus__extr.
Proof.
  unfold plus, plus__extr.
  ie_hnf.
  fix IHn 3.
  intros n n' Hnn'.
  ie_simpl.
  ie_hnf.
  intros m m' Hmm'.
  ie_simpl.
  pattern n at 1, n' at 1.
  ie_destruct_nat Hnn'; ie_simpl.

  (* O *)
  ie_easy.

  (* S *)
  apply S__simul.
  ie_apply; ie_easy.
Qed.
```

La preuve commence par déplier les définitions de `plus` et de `plus__extr`. Face à une conclusion de la forme `simul f f'`, où `f` est une fonction, on utilise la tactique `ie_hnf` qui se charge de transformer la conclusion afin de la faire commencer par des quantifications par variables hermétiques (c'est-à-dire de la forme `forall x x', ie_pred x x' -> ...`), puis on introduit ces variables avec `intros`, et on réduit autant que possible le terme mini-ML avec `ie_simpl` (voir section 4.4.2). L'introduction des variables

hermétiques correspondant à l'argument décroissant de `plus` est précédée d'un appel à la tactique `fix`, qui indique la construction d'un point fixe dans le terme de preuve. Lorsque le filtrage sur l'argument décroissant arrive en tête de réduction, on procède à une analyse par cas avec `pattern` et `ie_destruct_nat` (voir section 4.4.3), puis on réduit le terme mini-ML dans chaque sous-but. Le terme Coq est également réduit en parallèle, implicitement. Le cas `O` du filtrage revient à prouver `simul m m'` en ayant `ie_pred m m'` en hypothèse, ce qui se résout en utilisant le lemme `ie_compat` via la tactique `ie_easy`. Le cas `S` revient à prouver (on a replié les définitions de `plus` et de `plus__extr` par souci de clarté) :

```
simul (S (plus n m)) (TConstr 1 (plus__extr @ n' @ m' :: nil))
```

On fait alors disparaître le constructeur de tête avec le lemme `S__simul` (un lemme similaire est créé au préalable pour chaque constructeur), puis le but est décomposé en deux sous-buts avec `ie_apply`, un dont la conclusion est :

```
simul (plus n) (plus__extr @ n')
```

qui se résout par un appel (récursif) à l'hypothèse introduite par `fix`, et un autre dont la conclusion est la même que dans le cas `O`, et se résout de la même manière.

Ces techniques de preuve fonctionnent au moins pour la fonction d'Ackermann (points fixes imbriqués), la multiplication (utilisant une définition préalable, l'addition), les fonctions `length` et `map` sur les listes polymorphes, la fonction `test1` de la page 86, et même la fonction `double` de la page 81, qui manipule des sous-termes logiques. Elles ne fonctionnent pas avec le `pred` de la page 81 : plus précisément, la tactique `pattern` échoue, probablement à cause des types dépendants. Il est néanmoins possible d'achever la preuve manuellement.

4.5 Bilan

4.5.1 Contributions

Dans ce chapitre, nous avons exposé le principe d'une extraction directe à partir de Coq visant un langage entièrement formalisé. Notre langage source est le CIC implanté dans Coq, sans modification ni modélisation. Nous avons réalisé sous la forme d'un greffon une nouvelle commande vernaculaire d'extraction qui génère des termes sous une forme exploitable au niveau logique. Bien que cette extraction ne soit pas elle-même prouvée, nous avons présenté un cadre de certification des programmes extraits au sein du système Coq.

Le travail présenté ici est le résultat de très longues expérimentations. Il s'inspire à l'origine de l'étude sémantique de [33]. Toutefois, P. Letouzey avait réalisé cette étude en faisant notamment les deux hypothèses simplificatrices suivantes concernant CIC : les

promotions entre sortes sont explicites et deux termes mini-ML convertibles sont égaux. Ces deux hypothèses changent significativement CIC. Nous pensons que la première pourrait être implantée au sein de Coq et passer inaperçue pour l'utilisateur en ajoutant une phase d'insertion automatique de transtypages (*casts*) lors du pré-typage. Nous avons déjà mentionné cette idée comme aide possible pour la résolution du problème du polymorphisme de sortes mentionné à la section 3.4. La seconde hypothèse est encore plus envahissante et revient à intégrer une partie de la sémantique de mini-ML à CIC. Nous avons plutôt choisi de travailler avec Coq sans chercher à changer la théorie sous-jacente.

Nous avons proposé une notion de correction des termes extraits à travers une relation de simulation entre des termes Coq et des termes mini-ML. Cette relation repose sur des prédicats dits d'extractibilité reliant les termes Coq à des *valeurs* mini-ML, prédicats eux-mêmes combinés à une sémantique opérationnelle de mini-ML pour donner une notion s'appliquant à tous les *termes* mini-ML. Seuls les prédicats d'extractibilité sont spécifiques à chaque type et nécessitent des définitions dédiées. Nous pensons qu'ils peuvent être automatiquement générés pour tous les types ML.

Nous avons également exposé notre approche pour prouver la simulation entre un terme Coq et son extrait. Pour cela, nous reposons sur le système de tactiques. Nous avons présenté une démarche générale pour mener ces preuves, et nous avons défini de nouvelles tactiques pour manipuler les termes mini-ML, et en particulier les réduire. Nous avons cherché à simplifier au maximum ces preuves, dans un but d'automatisation.

Le développement concret fait environ 2 000 lignes de Coq et 300 lignes d'OCaml.

4.5.2 Limitations

Notre extraction remplace les sous-termes logiques par une constante inerte □, inexistante dans le modèle de notre langage cible — le εML de Z. Dargaye — pour laquelle nous n'avons pas donné de concrétisation. Nous avons vu au chapitre 3 que cette constante nécessitait des règles de réduction supplémentaires qui ne font pas partie de notre mini-ML. Du point de vue de l'extraction interne, cela se traduit par l'impossibilité de certifier des programmes nécessitant ces règles, comme le `boum` de la section 3.4. La concrétisation de □ pourrait se faire entièrement dans le cadre de mini-ML en ajoutant des tests dynamiques, mais une implantation efficace nécessiterait de descendre à un niveau plus bas dans la chaîne de compilation. Malgré l'absence de concrétisation de □, nous pouvons quand même traiter certains programmes avec parties logiques.

Nous pensons que notre approche permettrait de certifier automatiquement tous les termes extraits dont le terme source correspond à du ML typé. Dans l'état actuel de nos travaux, nous savons que les types dépendants ne peuvent pas être traités automatiquement, mais cela n'exclut pas toute amélioration future. En outre, même si nous n'avons pas produit de méthode générique pour certifier automatiquement *tous* les termes, des preuves manuelles sont toujours possibles, et les tactiques déjà élaborées en diminuent grandement la complexité.

4.5.3 Conclusion

Le choix d'utiliser l'implantation concrète de Coq a rendu possibles de nombreuses expérimentations, mais difficile tout raisonnement formel. Même si notre approche, qui au final repose sur une validation *a posteriori* par le noyau de Coq, ne nécessite pas de certification elle-même, il pourrait être souhaitable de formaliser précisément ce à quoi on peut s'attendre.

Tous les travaux exposés ici ont été réalisés en l'absence de formalisation précise et fidèle de CIC : le noyau de Coq a été considéré comme une boîte noire. Nous pensons néanmoins qu'une telle formalisation serait utile pour délimiter précisément le champ d'application de l'extraction interne. $\mathrm{CIC}_{\mathrm{B}}$ — la variante de CIC formalisée au chapitre 3 — s'éloigne trop de Coq pour cela ; nous allons présenter dans le chapitre 5 quelques idées pour une nouvelle formalisation de CIC.

5 Perspectives

Le travail de formalisation de l'extraction théorique autour du développement de B. Barras est significativement avancé, même s'il reste encore quelques lacunes. Toutefois, nous sommes encore loin d'une implantation pratique qui pourrait supplanter celle actuellement présente dans Coq, et permettre ainsi l'auto-génération d'un vérificateur autonome.

Pour atteindre ce but, nous pensons qu'une nouvelle formalisation de CIC, pensée dès le départ dans une optique d'interfaçage avec Coq, est nécessaire. Indépendamment de l'extraction, ce nouveau développement pourrait également être utilisé à des fins d'introspection, et fournir une réification universelle.

Nous allons présenter un hypothétique langage, $\mathrm{CIC_X}$, conçu dans ce but. Nous motivons nos choix en nous appuyant sur l'implantation existante de Coq, ainsi que sur l'expérience tirée de la formalisation de $\mathrm{CIC_B}$.

5.1 Vers une formalisation plus proche de Coq

La principale différence théorique de $\mathrm{CIC_X}$ par rapport à $\mathrm{CIC_B}$ est l'abandon des marques pour le typage des points fixes. Coq assure la normalisation des points fixes grâce à une condition de garde syntaxique : dans le corps d'un point fixe, tous les appels récursifs doivent être faits sur un sous-terme d'un argument inductif désigné. En ce qui concerne l'extraction, le choix entre marques et condition de garde syntaxique n'a pas d'importance ; seule la normalisation compte, et elle peut être (et l'est déjà dans $\mathrm{CIC_B}$) admise d'un point de vue logique sous la forme d'un axiome. La condition de garde, quant à elle, peut être modélisée par un oracle. C'est l'approche que nous adoptons pour $\mathrm{CIC_X}$.

Syntaxe Contrairement à $\mathrm{CIC_B}$, on abandonne la syntaxe à deux niveaux et la notion d'opérateur. Sur le même modèle que `name`, soit `identifier` un ensemble d'*identifiants*, c'est-à-dire un type de données quelconque, infini, dénombrable et dont l'égalité est décidable. Nous distinguons noms et identifiants par leur usage : un nom n'est pas significatif, et utilisé uniquement à des fins cosmétiques lors des interactions, alors qu'un identifiant est significatif et peut par exemple être sujet à des contraintes d'unicité. Les deux concepts étaient confondus dans $\mathrm{CIC_B}$, mais une distinction existe dans Coq.

Définition 5.1 (type `term`)**.** Les *termes* sont définis par la grammaire suivante :

$$\begin{aligned} T, U : \texttt{term} \quad ::= \quad & s \mid \mathcal{G}(\alpha) \mid \natural i \\ & \mid \forall x : T_1.T_2 \mid \lambda x : T_1.T_2 \mid T_1\ T_2 \\ & \mid \mathsf{let}\ x : T_1 = T_2\ \mathsf{in}\ T_3 \\ & \mid \exists(\vec{x}, \vec{T}) \mid \vec{T} \mid \pi_i(T) \\ & \mid \mathcal{I}(\alpha, i, T_1, T_2) \mid \mathcal{C}(\alpha, i_1, i_2, T_1, T_1) \\ & \mid \mathcal{M}(T_1, T_2, \vec{U}) \mid \mathcal{F}(\alpha, i, T_1, T_2) \end{aligned}$$

où $s \in \texttt{sort}$, $\alpha \in \texttt{identifier}$, $x \in \texttt{name}$ et $i, i_1, i_2 \in \mathbb{N}$.

PTS, constantes À l'instar de Coq et de CIC_B, CIC_X possède des sortes (les mêmes) pour typer les types, ainsi que des constantes globales nommées. Contrairement à CIC_B, les constantes n'ont pas de paramètre particulier, ce qui est aussi le cas de Coq (nous ignorons le polymorphisme de sorte). Les variables sont représentées avec des indices de de Bruijn, et le produit dépendant, l'abstraction et l'application reste inchangés. Par rapport à CIC_B, nous ajoutons les définitions locales (présentes dans Coq).

Sommes dépendantes CIC_X possède des sommes dépendantes n-aires primitives, et de quoi construire des n-uplets et effectuer des projections. Cette construction est présente dans CIC_B mais pas dans Coq. Ce choix va de pair avec celui de restreindre le nombre de paramètres et d'arguments des inductifs, constructeurs et points fixes. Comme pour les autres lieurs, il est possible de nommer les variables liées dans les sommes dépendantes.

Ces sommes dépendantes primitives ouvrent la voie à une optimisation conceptuellement simple des termes extraits : la suppression pure et simple (au lieu du remplacement par $\square$) des composantes logiques d'un n-uplet. En combinant cela avec une décurryfication systématique comme dans [16], on pourrait obtenir des termes extraits comparables à ceux de l'extraction implantée par P. Letouzey au sein de Coq en termes de « vestiges » logiques. Cependant, des précautions sont à prendre avec le sous-typage de `Prop` en `Type`, qui pose un problème similaire à celui du polymorphisme de sorte mentionné à la section 3.4.

Types inductifs Tout comme dans Coq et CIC_B, les inductifs sont déclarés globalement au sein de la signature. Chaque déclaration, qui peut contenir plusieurs types mutuellement inductifs, possède un identifiant. Un inductif est référencé par l'identifiant de sa déclaration et son indice. De même, chaque constructeur est référencé par l'identifiant de sa déclaration, l'indice de son type au sein de la déclaration, et son propre indice au sein du type. Les inductifs et les constructeurs ont deux sous-termes : un paramètre et un argument. Dans Coq, les inductifs et les constructeurs peuvent avoir un nombre quelconque de paramètres et d'arguments, et cela peut se traduire en CIC_X grâce aux sommes dépendantes primitives.

Les déclarations elles-mêmes (et leur typage) ne devraient pas différer significativement de celles de $\mathrm{CIC_B}$ et de Coq.

Filtrage Le filtrage possède en sous-termes le terme filtré, le prédicat de retour et la liste des branches. Chaque branche est indexée par son constructeur et est une fonction prenant en argument l'argument du constructeur.

Points fixes Tous les points fixes sont déclarés globalement au sein de la signature. Chaque déclaration, qui peut contenir plusieurs fonctions mutuellement récursives, possède un identifiant. Un point fixe particulier est référencé par l'identifiant de sa déclaration et son indice, et est obligatoirement appliqué à son paramètre et son argument décroissant. Il est toujours possible de désigner un point fixe non appliqué en procédant à une *η-expansion*. Ce choix de points fixes globaux et nommés diffère de Coq et de $\mathrm{CIC_B}$, où les points fixes, possiblement locaux, sont anonymes. Nous le justifions par les raisons suivantes :

- la règle originale de réduction des points fixes impose leur dépliage systématique, alors que cela n'est souvent pas souhaitable. D'ailleurs, cela se manifeste dans la tactique `simpl` de Coq, qui remplace le but courant par un terme convertible. Dans la plupart des cas, le résultat est obtenu par une série de réductions dans le sens direct sans déplier de définition. Toutefois, les définitions de points fixes sont traitées spécialement : si une réduction a lieu (*i.e.* l'argument commence par un constructeur), alors la définition est dépliée, la réduction est effectuée, puis la définition est repliée ; sinon, rien n'est fait. Par exemple, une exécution de `simpl` peut se détailler comme suit :

$$
\begin{array}{lll}
\texttt{plus (S n) m} & \to_\delta & \texttt{(fix plus n := ...) (S n) m} \\
 & \to_\iota & \left\{ \begin{array}{l} \texttt{match S n with} \\ \texttt{| O => m} \\ \texttt{| S p => S ((fix plus n := ...) p m)} \end{array} \right. \\
 & \to_\iota & \texttt{S ((fix plus n := ...) n m)} \\
 & \leftarrow_\delta & \texttt{S (plus n m)}
\end{array}
$$

  Les trois premières étapes sont des réductions dans le sens direct, alors que la dernière est une réduction en sens inverse ;
- cette représentation rapproche le traitement des types inductifs et des points fixes, et ouvre la voie aux déclarations simultanées de types inductifs et de points fixes mutuels, appelées *inductifs-récursifs* dans la littérature.

Il est possible de rendre globaux des points fixes locaux via une technique de *λ-lifting*. Les points fixes sont également globaux et nommés dans Matita [1].

Les déclarations de points fixes sont typées comme le seraient les points fixes anonymes, en faisant appel à l'oracle mentionné plus tôt pour la condition de garde.

Gestion de l'environnement Selon la terminologie employée dans Coq, l'« environnement » comprend aussi bien l'environnement global (ce qui est appelé « signature » dans $\mathrm{CIC_B}$: les déclarations de constantes et de types inductifs) que l'environnement local (ce qui est appelé « environnement » dans $\mathrm{CIC_B}$: ce que référencent les indices de de Bruijn). Nous séparons de plus la signature en deux composantes : d'un côté, nous mettons les inductifs et les points fixes ; de l'autre, les constantes.

Définition 5.2 (type `decl`)**.** Une *déclaration* est un triplet $\langle x, T_1, T_2^\star \rangle$ où x est un nom ou un identifiant, T_1 un terme appelé *type*, et $T_2^\star$ un terme optionnel appelé *corps*.

Définition 5.3 (type `heap`)**.** Un *tas* est un ensemble de déclarations inductives et de déclarations de points fixes :

$$\begin{array}{rcl} H : \texttt{heap} & ::= & (I \mid F)^* \\ I : \texttt{ind\_pack} & ::= & \langle \alpha, i^* \rangle \\ i : \texttt{one\_ind} & ::= & \langle x, P, A, r, s, c^\star \rangle \\ F : \texttt{fix\_pack} & ::= & \langle \alpha, f^* \rangle \\ f : \texttt{one\_fix} & ::= & \langle x, P, A, t \rangle \end{array}$$

où $\alpha \in \texttt{identifier}$, $x \in \texttt{name}$, $P, A, t \in \texttt{term}$ et $r, s \in \texttt{sort}$.

Définition 5.4 (type `env`)**.** Un *environnement* est un triplet $\Gamma = \langle H, S, C \rangle$ où H est un tas, S un ensemble de déclarations avec identifiants appelé *signature*, et C une liste de déclarations avec noms appelée *contexte*.

L'union du tas et de la signature forme ce qui est appelé « signature » dans $\mathrm{CIC_B}$. Nous les avons séparés pour des raisons esthétiques : ainsi, il y a une analogie entre signature et contexte (ce sont tous les deux des ensembles de `decl` indexés différemment), alors que le tas correspond à un ensemble de primitives dont le rôle est fondamentalement différent.

Gestion des entités globales La gestion du nommage des entités globales est différente de celle de $\mathrm{CIC_B}$: par exemple, il y a un identifiant par déclaration d'inductif et non pas un identifiant par type inductif. Les constructeurs sont référencés par l'identifiant de leur déclaration et leurs indices. Dans Coq, les déclarations n'ont pas de nom explicite mais existent sous la forme d'une entité abstraite qui est référencée par les termes. $\mathrm{CIC_X}$ se rapproche ainsi de Coq, mais nous proposons une interface légèrement différente. Ce qui suit prend en exemple les constructeurs, mais s'applique tout aussi bien aux inductifs et aux points fixes.

Dans Coq, chaque constructeur possède un identifiant au même titre que les constantes, et l'espace de nommage est partagé avec elles. Les constructeurs ont une existence autonome, et l'instanciation de leurs paramètres et arguments se fait via ce qui semble être l'application traditionnelle du λ-calcul. Cela a pour conséquence que les

constructeurs donnent l'illusion d'être des constantes alors qu'en réalité plusieurs sens sont donnés à l'application. En particulier, une expression d'un type fonctionnel n'est pas forcément une fonction dans Coq, et cela rend pénible l'énoncé de certains lemmes tels que 3.22. Nous proposons plutôt de forcer les constructeurs à être toujours accompagnés de leurs paramètres et arguments au niveau du noyau. Ainsi, l'application conserve son sens « pur » du λ-calcul.

Réduction et conversion Les règles de réduction de $\mathrm{CIC_X}$ sont les mêmes que pour $\mathrm{CIC_B}$ et Coq. En définissant la relation de convertibilité comme la simple clôture réflexive, symétrique et transitive de la réduction, on obtient une relation légèrement différente, à cause du traitement des points fixes : deux fonctions récursives différentes ayant le même corps ne sont plus convertibles. Cette différence est présente dans Matita et mentionnée dans [1], mais ne semble pas problématique en pratique.

Sous-typage En l'absence de marques, la seule source de sous-typage est l'inclusion des sortes. Dans $\mathrm{CIC_B}$, le produit dépendant est contravariant à gauche, mais ce n'est pas le cas dans Coq car cette propriété n'est pas validée par les modèles ensemblistes. Ce choix semble avoir peu d'impact en pratique, et nous adoptons celui de Coq.

Typage Nous ne prévoyons pas de difficulté particulière pour l'adaptation de la métathéorie du typage à $\mathrm{CIC_X}$.

Extraction La nouvelle formalisation est l'occasion de pallier certaines limitations mentionnées à la section 3.5.3. En suivant le modèle de la section 3.2, on peut définir un langage $\mathrm{CIC_{X'}}$ et considérer une extraction de $\mathrm{CIC_X}$ vers $\mathrm{CIC_{X'}}$. À cause de la globalité des points fixes, il est nécessaire de définir l'extraction sur l'environnement ; au lieu d'un jugement de la forme :

$$\langle\Sigma\rangle\Gamma \vdash t \to_{\mathcal{E}} t'$$

l'étude porterait plutôt sur un jugement de la forme :

$$\langle \Gamma \mid t\rangle \to_{\mathcal{E}} \langle \Gamma' \mid t'\rangle$$

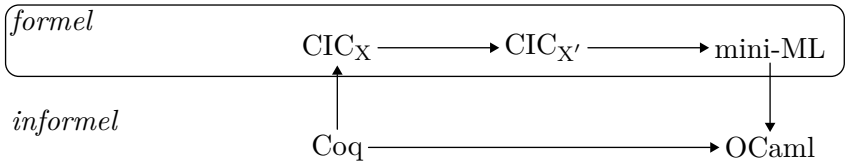
où Γ' représente l'environnement Γ (complet, au sens de $\mathrm{CIC_X}$) où les termes ont été extraits. La notion de clôture utilisée pour les équivalents des théorèmes 3.29 et 3.30 porterait alors sur le couple de l'environnement et du terme et pourrait tenir compte des constantes et variables avec un corps.

5.2 Vers un Coq plus proche de la formalisation

$\mathrm{CIC_X}$ est conçu de sorte que la transformation d'un jugement de typage $\Gamma_0 \vdash t_0 : T_0$ de Coq en un jugement $\Gamma \vdash t : T$ de $\mathrm{CIC_X}$ soit réalisable en pratique. Cette transformation

peut servir de base à un vérificateur autonome de conception radicalement différente. Étant donné un environnement Σ (en Coq), un greffon (comme Xcoq) se chargerait de la convertir en un environnement Σ' (en $\mathrm{CIC_X}$) dans un format que l'extrait informel du vérificateur de types de $\mathrm{CIC_X}$ accepte.

Remplacer l'extraction informelle par une extraction extraite nécessiterait de plus une transformation de $\mathrm{CIC_{X'}}$ vers mini-ML. Ce chemin de Coq vers mini-ML semble plus prometteur que l'extraction interne.



Nous n'avons présenté $\mathrm{CIC_X}$ que du point de vue « noyau », sans considérer l'interface avec l'utilisateur. Remplacer naïvement CIC par $\mathrm{CIC_X}$ dans Coq produirait un système *a priori* différent, mais nous pensons qu'il est possible de fournir une expérience comparable pour l'utilisateur.

Une différence entre CIC et $\mathrm{CIC_X}$ est la disparition des points fixes locaux et anonymes. Nous pensons qu'il est possible de continuer à maintenir l'illusion de points fixes anonymes pour l'utilisateur, en fournissant un mécanisme hors du noyau qui se chargerait de construire automatiquement les déclarations globales (au sein du tas) en abstrayant le corps par rapport aux variables libres et en générant un identifiant unique. La commande vernaculaire `Fixpoint` ferait également appel à ce mécanisme, et construirait de plus une constante (au sein de la signature) dont le corps serait une fonction représentant le point fixe après *η-expansion*.

Ce mécanisme de déclaration automatique peut être poussé plus loin et appliqué aux inductifs : la commande `Inductive`, outre l'enregistrement dans le tas, construirait des constantes pour chaque constructeur et chaque type de la déclaration. L'abstraction automatique par rapport aux variables libres fournirait également des inductifs locaux à peu de frais.

Dans cette optique, tous les identifiants du tas sont générés et gérés par le système (mais hors du noyau), et l'utilisateur ne nommerait que les constantes de la signature. Même si, concrètement, les noms de la signature peuvent être utilisés comme suggestions, les identifiants utilisés pour indexer le tas doivent être vus comme des pointeurs, hors du contrôle de l'utilisateur ordinaire. Le traitement des identifiants globaux choisis par l'utilisateur serait ainsi plus uniforme.

5.3 Bilan

Une formalisation complète de $\mathrm{CIC_X}$, et l'implantation de la traduction automatique de Coq vers $\mathrm{CIC_X}$ ne semblent pas insurmontables mais nécessitent du temps, et peuvent

être l'objet de travaux futurs. Nous pensons qu'elles sont nécessaires pour une formalisation plus précise de l'extraction. Elles pourraient également servir de support pour d'autres aspects de CIC indépendants de l'extraction, tels que la condition de garde, les co-inductifs, les modules, le polymorphisme de sorte... tout en gardant à l'esprit l'aspect pratique de l'implantation de Coq.

Cette thèse s'est principalement concentrée sur le côté « langage source » de l'extraction ; le côté « langage cible » a été peu exploré. En particulier, l'intégration de fonctions certifiées extraites au sein de programmes plus complexes contenant du code non certifié mériterait une étude approfondie.

Bibliographie

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A compact kernel for the calculus of inductive constructions. *Journal Sadhana*, 34 :71–144, 2009.

[2] David Aspinall. Proof General : A generic tool for proof development. In *Proceedings of TACAS'2000*, volume 1785. Lecture Notes in Computer Science, 2000. Software available at `http://proofgeneral.inf.ed.ac.uk`.

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

[4] Henk Barendregt. Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen, 1991. In Handbook of Logic in Computer Science, Vol II.

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

[6] Bruno Barras. Sets in Coq, Coq in Sets. *Journal of Formalized Reasoning*, 3(1) :29–48, 2010.

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

[8] 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.

[9] Stefan 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.

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

[11] Jacek Chrząszcz. Implementing modules in the system Coq. In *16th International Conference on Theorem Proving in Higher Order Logics*, University of Rome III, September 2003.

[12] Alonzo Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 33(2) :pp. 346–366, 1932.

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

[14] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers : A comparison with Astrée, invited paper. In He Jifeng and J. Sanders, editors, *Proc. First IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE'07*, pages 3–17, Shanghai, China, 6–8 June 2007. IEEE Computer Society Press, Los Alamitos, California, United States.

[15] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

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

[17] Zaynah Dargaye. *Vérification formelle d'un compilateur pour langages fonctionnels.* Thèse de doctorat, Université Paris Diderot-Paris 7, July 2009.

[18] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *INDAG. MATH*, 34 :381–392, 1972.

[19] Jean-Christophe Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[20] Herman Geuvers. Inconsistency of classical logic in type theory, November 2001.

[21] Eduardo Giménez. An application of co-inductive types in Coq : verification of the Alternating Bit Protocol. In *Workshop on Types for Proofs and Programs*, number 1158 in LNCS, pages 135–152. Springer-Verlag, 1995.

[22] Stéphane Glondu. *Garantie formelle de correction pour l'extraction Coq.* Master's thesis, Université Paris Diderot-Paris 7, 2007.

[23] Stéphane Glondu. Extraction certifiée dans Coq-en-Coq. *Journées Francophones des Langages Applicatifs (JFLA'09). INRIA*, 2009.

[24] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

[25] Benjamin Grégoire and Jorge Sacchini. On Strong Normalization of the Calculus of Constructions with Type-Based Termination. In *Logic for Programming, Artificial Intelligence, and Reasoning : 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010, Proceedings 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 333–347, Yogyakarta Indonésie, November 2010. The original publication is available at www.springerlink.com.

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

[27] William 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.

[28] Richard Kelsey, William Clinger, and Jonathan Rees. *Revised*^{5} *Report on the Algorithmic Language Scheme*, 1998. Available at `http://schemers.org/`.

[29] Stephen C. Kleene. *Introduction to Metamathematics.* North-Holland, Amsterdam, 1952.

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

[31] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.

[32] Xavier Leroy, Jérôme Vouillon, Damien Doliguez, Jacques Garrigue, and Didier Rémy. *The OCaml system – release 3.12*, July 2011. Available at `http://caml.inria.fr/`.

[33] Pierre Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq.* Thèse de doctorat, Université Paris-Sud, July 2004.

[34] Zhaohui Luo. ECC, an extended calculus of constructions. In *LICS*, pages 386–395. IEEE Computer Society, 1989.

[35] George C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.

[36] Christine 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.

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

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

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

[40] Lawrence C. Paulson. *Logic and Computation : Interactive Proof with Cambridge LCF.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[41] Simon Peyton Jones et al. *Haskell 98, A Non-strict, Purely Functional Language*, 1999. Available at `http://haskell.org/`.

[42] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.

[43] Élie Soubiran. *Modular development of theories and name-space management for the Coq proof assistant*. PhD thesis, École Polytechnique, 2010.

[44] Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2006.

[45] Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[46] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.3*, October 2010. Available at `http://coq.inria.fr/`.

[47] Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives*. Thèse de doctorat, Université Paris 7, 1994.

[48] Freek Wiedijk. Comparing mathematical provers. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2003.

Index

Résumé

L'assistant de preuve Coq permet de s'assurer mécaniquement de la correction de chaque étape de raisonnement dans une preuve. Ce système peut également servir au développement de programmes certifiés. En effet, Coq utilise en interne un langage typé dérivé du λ-calcul, le calcul des constructions inductives (CIC). Ce langage est directement utilisable pour programmer, et un mécanisme — l'extraction — permet de traduire les programmes CIC vers des langages à plus large audience tels qu'OCaml, Haskell ou Scheme.

L'extraction n'est pas un simple changement de syntaxe : CIC dispose d'un système de types très riche, mais en contrepartie, des entités purement logiques peuvent apparaître dans les programmes et impacter leurs performances. L'extraction se charge également d'effacer ces parties logiques.

Dans cette thèse, nous nous attaquons à la certification de l'extraction elle-même. Nous avons prouvé sa correction dans le cadre d'une formalisation entière de Coq en Coq. Cette formalisation ne correspond pas exactement au CIC implanté dans Coq, mais nous avons tout de même réalisé notre étude avec l'implantation concrète de Coq en tête. Nous proposons également une nouvelle méthode de certification des programmes extraits, dans le cadre concret du système Coq existant.

Mots-clés preuve de programmes, programmation fonctionnelle, extraction, théorie des types, Curry-Howard, calcul des constructions inductives, Coq en Coq

Abstract

The Coq proof assistant mechanically checks the consistency of the logical reasoning in a proof. It can also be used to develop certified programs. Indeed, Coq uses internally a typed language derived from λ-calculus, the calculus of inductive constructions (CIC). This language can be directly used by a programmer, and a procedure — extraction — allows one to translate CIC programs into more widely used languages such as OCaml, Haskell or Scheme.

Extraction is not a mere syntax change: the type system of CIC is very rich, but purely logical entities can appear inside programs, impacting their performance. Extraction erases these logical artefacts as well.

In this thesis, we tackle certification of the extraction itself. We have proved its correction in the context of a full formalization of Coq in Coq. Even though this formalization is not exactly Coq, we worked on it with the concrete implementation of Coq in mind. We also propose a new way to certify extracted programs, in the concrete setting of the existing Coq system.

Keywords proof of programs, functional programming, extraction, type theory, Curry-Howard, calculus of inductive constructions, Coq in Coq