Normalisation par Évaluation en Coq
Rapport de Stage de Master 2
Master Parisien de Recherche en Informatique

François Garillot
francois.garillot©ens.fr

Sous la direction de Benjamin Werner*
mars-août 2007

Résumé

On caractérise formellement la normalisation par évaluation des termes du λ-calcul simplement typé à l'aide d'une formalisation en Coq, et prouvons la correction d'un algorithme de normalisation indexé par les types. Cet algorithme est implémenté dans le cadre d'une formalisation à l'aide de variables nommées, réalisant une implémentation de la substitution simultanée définie par Stoughton. On établit ensuite un exemple d'adaptation de cette formalisation pour la représentation de termes du λ-calcul en syntaxe abstraite d'ordre supérieur.

*Équipe LOGICAL, LIX, École Polytechnique

Table des matières

Introduction

La normalisation par évaluation est une discipline usuellement évoquée dans un cadre impliquant la réduction de termes du λ-calcul : plutôt que normaliser un terme sur la base de règles de réduction, on définit une fonction de normalisation qui envoie un terme vers sa forme normale. Si Martin-Löf suggérait déjà dans [**?**] la théorie des types intuitionniste comme métalangage, le premier développement algorithmique de cette fonction en elle-même est dû à [BS91].

La motivation originelle de l'algorithme de normalisation par évaluation était essentiellement pratique : il s'agissait d'obtenir des représentation des objets de preuve du système de preuve MinLog, développé à LMU. C'est ce qui a conduit Berger et al. à décrire leur travail comme un «inverse de la fonction d'évaluation».

En fait, la normalisation par évaluation est un processus qui utilise le mécanisme d'évaluation d'un métalangage pour normaliser des termes, typiquement ceux du λ-calcul. Nous souhaitons ici en fournir une formalisation en Coq, prouvant la correction d'un algorithme de normalisation, et fournissant un formalisme sur lequel il est possible de parler des formes normales de termes du lambda calcul représentés à l'aide d'une syntaxe d'ordre supérieur.

Dans la première partie, nous expliciterons l'algorithmique de la normalisation par évaluation pour le λ-calcul simplement typé, en mettant en évidence un argument de correction schématisant la preuve qui suivra, et la nécessité de génération de variables fraîches. Nous discutons ensuite du choix d'une représentation de variables appropriée, et des conséquences quand à notre formalisation. La seconde partie nous permettra d'évoquer l'usage des tactiques de réflection booléenne, fer de lance de nos scripts de preuve en Coq.

Dans les trois parties qui suivront, nous donnons la traduction de notre choix de sémantique et d'implémentation par rapport à notre formalisation, avant d'évoquer la preuve en elle-même, en reprenant une partie du contenu de [WG07], écrit durant ce stage.

Enfin, nous évoquerons de façon intuitive l'usage possible de cette normalisation pour la description de termes exprimés en syntaxe abstraite d'ordre supérieur, et les développements futurs qui pourraient en découler.

1 Normalisation par Évaluation

1.1 Algorithme

Considérons d'abord le versant sémantique de la normalisation par évaluation : à l'aide d'une fonction d'interprétation $[[.]]$, les termes sont injectés dans le métalangage. Une fonction de réification $\downarrow$, aussi appelée *inverse de la fonctionnelle d'évaluation*, sert à récupérer les termes à partir de leur sémantique.

Les deux propriétés essentielles de ces fonctions sont la *correction* :

$$r \to s \Rightarrow [[r]]_\varsigma = [[s]]_\varsigma$$

et la *reproduction*

$$r \text{ en forme normale } \Rightarrow \downarrow [[r]]_\uparrow = r$$

pour $\uparrow$ une valuation particulière.

Les types du λ-calcul sont de la forme $\tau ::= b|\tau_1 \to \tau_2$. Une interprétation ensembliste naturelle serait donc d'associer à chaque type de base b un ensemble, et au type fonctionnel l'espace fonctionnel ensembliste associé, i.e., $[[\tau_1 \to \tau_2]] = [[\tau_1]] \to [[\tau_2]]$.

Supposons ensuite que l'on nomme VAR, LAM et APP les constructeurs usuels des termes du λ-calcul, éléments de Λ. Alors, pour tout type τ, il devient assez naturel de construire la sémantique d'un terme $m : \tau$ de la façon suivante :

$$\begin{array}{rcl} [[\text{VAR}(x)]]_\rho & = & \rho(x) \\ [[\text{LAM}(x^\tau, m_0)]]_\rho & = & \lambda a^{[[\rho]]}.\, [[m_0]]_{\rho[x\mapsto a]} \\ [[\text{APP}(m_1, m_2)]]_\rho & = & [[m_1]]\, \rho([[m_2]]_\rho) \end{array}$$

Il est aisé de vérifier que ce modèle vérifie la propriété de correction, i.e. si $m \leftrightarrow_{\beta\eta} m'$, $[[m]]_\varsigma = [[m']]_\varsigma$ pour toute valuation ς. On choisit alors le modèle qui associe aux types de base b l'ensemble Λ pour tout b. Il devient alors simple de construire une fonction $\mathrm{nf}_\tau \in [[tau]] \to \Lambda$ telle que pour tout terme clos E en forme β-normale η-longue, $\mathrm{nf}_\tau([[E]]) = E$.

Si τ se décompose en $\tau = \tau_1 \to \ldots \to \tau_n \to b$, où $\forall i\ \tau_i = \tau_{i1} \to \ldots \to \tau_{im_i} \to b_i$, alors $E : \tau$ doit être de la forme $\lambda x_1 \ldots . \lambda x_n .\, [[E]]\, (\ldots (x_i E_1 \ldots E_{m_i}) \ldots)$ où chaque $E_j : \tau_{ij}$ est à nouveau en forme normale. Ceci nous amène à la définition inductive de nf_τ comme :

$$\begin{array}{rcl} \mathrm{nf}_\tau : f & \to & \text{LAM}(v_1, \ldots, \text{LAM}(v_n, \\ & & \quad f(\lambda a_1 \ldots . \lambda a_{m1}.\text{APP}(\ldots \text{APP}(\text{VAR}v_1, \mathrm{nf}_{\tau_{11}}(a_1)) \ldots, \mathrm{nf}_{\tau 1 m_n}(a_{mn}))) \\ & & \quad\ \vdots \\ & & \quad (\lambda a_1 \ldots . \lambda a_{mn}.\text{APP}(\ldots \text{APP}(\text{VAR}v_n, \mathrm{nf}_{\tau_{n1}}(a_1)) \ldots, \mathrm{nf}_{\tau n m_n}(a_{mn}))) \ )) \end{array}$$

On remarque que le développement de la forme normale de $[[E]]$ reflète la nature duale de τ, dont les constructeurs d'occurrence positive nous conduisent à faire une η-expansion des termes interprétés, ou objets, et dont les constructeurs d'occurrence négative nous conduisent à faire une η-expansion des méta-termes. Cela nous amène naturellement à une formulation mutuellement inductive de nf_τ, indexée par les types, à l'aide de deux familles de fonctions, *réifie*, $\downarrow^\tau : [[\tau]] \to \Lambda$ et *réfléchit*, $\uparrow^\tau : \Lambda \to [[\tau]]$.

$$\begin{array}{rcl} \downarrow^b l & = & l \\ \downarrow^{\tau_1 \to \tau_2} f & = & \text{LAM}(x^{\tau_1}, \downarrow^{\tau_2} (f(\uparrow^{\tau_1} \text{VAR}(x)))) \text{ pour } x \text{ variable fraiche} \\ \uparrow^b l & = & l \\ \uparrow^{\tau_1 \to \tau_2} l & = & \lambda a^{[[\tau_1]]}.\, \uparrow^{\tau_2} (\text{APP}(l, \downarrow^{\tau_1} a)) \end{array}$$

On vérifie bien que $\mathrm{nf}_\tau([[E]]) = \downarrow^\tau ([[E]]_\uparrow) = E$ pour E en forme β-normale η-longue, par induction sur E :

Cas : $\text{APP}(\text{VAR}(x)^{\rho\to\tau}, N^\rho)$

$$\downarrow^\tau ([[xN]]_\uparrow) = \uparrow^{\rho\to\tau} (x)([[N]]_\uparrow) = \uparrow^\tau (APP(\text{VAR}(x), \downarrow^\rho ([[N]]_\uparrow))) = APP(\text{VAR}(x), N)$$

Cas : $\textsc{lam}(y, N)$

$$\begin{aligned}
\downarrow^{\rho\to\sigma} ([[\textsc{lam}(y,N)]]_{\uparrow}) &= \textsc{lam}(x, \downarrow^{\sigma} ([[\textsc{lam}(y,N)]]_{\uparrow} (\uparrow^{\rho} \textsc{var}(x)))) \text{ pour } x \text{ fraiche} \\
&= \textsc{lam}(x, \downarrow^{\sigma} ([[N]]_{\uparrow[y\mapsto x]})) \\
&= \textsc{lam}(x, N_{[\textsc{var}(y)\mapsto\textsc{var}(x)]}) \\
&=_{\alpha} \textsc{lam}(y, N)
\end{aligned}$$

La conjonction des deux propriétés de correction et de représentation nous prouve donc que nf_{τ} est une fonction de normalisation.

1.2 Choix d'implémentation pour une formalisation rigoureuse

L'algorithme présenté ci-dessus algorithme est attribué à [BS91], ou plutôt, si l'on considère l'implémentation la plus proche de le forme simplifiée qui en est donnée, de l'extension de ce travail faite par Danvy pour l'évaluation partielle dirigée par les types ([Dan96]).

En effet, par souci de simplicité, cette présentation de l'algorithme de Berger et al. fait apparaître une petite difficulté dans l'optique d'une formalisation rigoureuse : expliciter la mention «pour x variable fraîche».

Dans le cadre d'une simple implémentation de l'algorithme, où l'on a une compréhension opérationnelle de la condition de fraîcheur ainsi mise en évidence, ce n'est pas un problème : on dispose de structures tels que le `gensym` de Scheme pour la réaliser. C'est en l'espèce ce que font de nombreux développements basés sur cet algorithme, se plaçant dans un cadre où les variables du métalangage sont représentés par des noms ([Ves01], [Mog99], [Dan96], [ABDM03]).

Une réponse usuelle aux problèmes de capture de ce type est de modéliser les variables du métalangage à l'aide d'indices (ou de niveaux) de de Bruijn, solution que nous avions initialement envisagé lors de ce travail de stage. Cette solution est ici impossible : on remarquera que la fonction $\downarrow^{\tau_1\to\tau_2}$ introduit des métavariables sous l'application d'un terme de type $[[\tau_1 \to \tau_2]]$. La position structurelle de cette métavariable dans le méta-terme renvoyé par $\downarrow^{\tau_1\to\tau_2}$ est donc déterminée dynamiquement, donc l'*indice* de de Bruijn souhaité pour cette variable, reflet de cette position, ne saurait être déterminé dans la définition de cette fonction.

L'intuition tendrait alors à utiliser des *niveaux* de de Bruijn, en conservant en paramètre des fonctions $\downarrow^{\tau}$ et $\uparrow^{\tau}$ l'état du niveau de de Bruijn courant afin de s'assurer qu'à chaque appel récursif, on génère des indices de de Bruijn plus grands que ceux déjà présents dans le terme que l'on est en train de manipuler. C'est toutefois le problème dual qui se pose alors : l'ensemble de niveaux de de Bruijn généré par la pile d'appels issue d'un appel à $\uparrow^{\tau_1\to\tau_2}$ sera employé dans des trous de profondeur différents dans le méta-terme renvoyé par la réification. Le niveau de profondeur à respecter n'est donc pas un paramètre statique de $\uparrow^{\tau}$.

1.3 Comparaison des représentations de lieurs pour l'algorithme de normalisation

Dans le cadre d'une formalisation, la condition de fraîcheur de l'algorithme doit donc être explicitée. A notre connaissance, la littérature sur le sujet recèle trois pistes possibles.

La première consiste à résoudre le problème qui précède, selon la formalisation mise en évidence par Berger et al. ([BES98]). On se place dans le contexte de niveaux de de Bruijn, et à *retarder le lifting*. Pour cela, on représente nos méta-termes en les interprétant non plus dans Λ mais à l'aide d'une famille de termes,

$$\widehat{\Lambda} = \mathcal{N} \to \Lambda$$

avec $\mathbb{N} \subseteq \mathcal{N}$. L'idée est de considérer les méta-termes comme des fonctions associant à un entier k le λ-terme auquel ce méta-terme ressemblerait sous k lieurs. On en tire naturellement l'interprétation dénotationnelle des termes du langage objet.

Puis on définit les fonction *réifie* $\downarrow^\tau : [[\tau]] \to \widehat{\Lambda}$ et *réfléchit*, $\uparrow^\tau : \widehat{\Lambda} \to [[\tau]]$ de la façon suivante :

$$\begin{array}{rcl}
\downarrow^b l & = & l \\
\downarrow^{\tau_1 \to \tau_2} f & = & \lambda n.\textsc{lam}(\tau_1, \downarrow^{\tau_2} (f(\uparrow^{\tau_1} \lambda k.(\textsc{var} n))(n+1))) \text{ pour } x \text{ variable fraiche} \\
\uparrow^b l & = & l \\
\uparrow^{\tau_1 \to \tau_2} l & = & \lambda a^{[[\tau_1]]}.\ \uparrow^{\tau_2} (\lambda n.\textsc{app}(l(n), (\downarrow^{\tau_1} a)(n)))
\end{array}$$

La solution repose sur le fait que le *niveau* de de Bruijn introduit dans $\downarrow_\tau$ quel que soit son emplacement final, tout en assurant que les appels récursifs successifs, quels que soient leur emplacement, généreront des termes *liftés* par le successeur de cet entier, soit respectant la condition de fraîcheur initialement souhaitée.

Cet argument informel est développé de manière rigoureuse dans [BES98], et fournit une solution entièrement compositionnelle, qui a été adoptée dans de nombreuses extensions de cet algorithme, comme [AJ04], [BES03] ou [Fil99].

Une seconde solution consiste à revenir dans un cadre nommé et à user de langages supportant la spécification de contraintes de fraîcheur, basés sur la logique nominale de Pitts ([GP02]), comme dans [SPG03] ou [Pot07]. Ce choix nous est interdit dans le cadre d'une formalisation en Coq, mais il est à noter que la présence d'implémentations de normalisation par évaluation dans ces articles est justifiée dans [Pot07] de la façon suivante :

> Normalization by evaluation is an interesting benchmark because it makes non-trivial use of names and environments.

Une formalisation de la normalisation par évaluation à l'aide du module nominal d'Isabelle/HOL ([UBN07]) recèlerait donc un intérêt certain pour juger de l'apport d'un tel formalisme à un assistant de preuve.

La troisième solution, que nous avons choisie, est d'expliciter la génération de noms de manière concrète, et se plaçant dans un contexte nommé. [Fil01], bien que ne comportant pas d'implémentation concrète, fournit une alternative en explicitant un algorithme très similaire en se basant sur des monades. Toutefois, le lien entre cette formalisation et une

formalisation en Coq à l'aide de variables nommées parait relativement moins évident que la solution qui a été retenue en pratique.

En effet, la formalisation de C.Coquand ([Coq02]), réalisée en ALF, est assez singulière (voir partie 3.5) et se caractérise par une ré-implémentation complète du *gensym* des implémentations voisine, à l'aide d'une définition cofinie. Par ailleurs, nous placer dans un cadre nommé nous demande de formaliser précisément la notion d'α-conversion dans ce cadre, choix qui n'est pas anodin : la formalisation de la substitution sans capture de manière structurelle devient ardue, et la notion d'α-conversion difficile à formaliser. On trouvera une discussion et des références sur les formalisations possibles dans ([ACP+07]). [Sto88] fournit une méthode de formalisation à l'aide de substitution simultanée, jointe à une génération de variables définie également de manière cofinie, qui nous a permis de rendre maîtrisables les lemmes élémentaires liées aux lieurs dont nous avions besoin pour notre formalisation. On en trouvera les détails en partie 6.2. Il est à noter que à notre connaissance, ce présent travail est à ce jour la seule formalisation conduite dans un assistant de preuve à l'aide de cette méthode. De fait, elle ré-implémente la totalité des trois premières parties de ([Sto88]).

2 Réflection à petite échelle

2.1 Réflection booléenne

Ce travail a été effectué à l'aide de l'extension SSREFLECT, développée par Georges Gonthier pour la preuve du théorème des quatre couleurs[Gon05b]. Cette extension se compose d'un nouveau jeu de macro-commandes pour l'interpréteur de Coq, et d'une librairie permettant d'implémenter une réflection booléenne.

On trouvera une exposition de la mécanique derrière cette extension dans les travaux qui en font usage, notamment [GMR+07], mais qu'il nous soit permis de faire ici une ébauche de quelques uns de ses traits les plus marquants.

Les développements utilisant cette extension souhaitent tout d'abord se placer dans un domaine décidable. Sur celui-ci, la distinction intuitionniste, faite par Coq, entre les propositions logiques et les valeurs booléennes devient extrêmement encombrante.

Par ailleurs, les expressions ont une propriété de substitivité qui rend la réécriture, transformation reine dans cette extension de Coq, extrêmement confortable. L'écriture de termes à l'aide de booléens va donc toujours être privilégiée par rapport à l'utilisation de propositions logiques. Pour cela, on emploie une coercion `is_true` des booléens vers les propositions, qui peut également être vue comme une déclaration de sous-typage.

```
Coercion is_true (b :bool) := b = true.
```

On relie alors propositions et booléens par le prédicat inductif `reflect` suivant, qui exprime l'équivalence logique entre une proposition `P` et la coercion d'un booléen `b`.

```
Inductive reflect (P : Prop) : bool → Set :=
  | Reflect_true : P → reflect P true
  | Reflect_false : ~ P → reflect P false.
```

Les objets de ce type `reflect` sont à la base du mécanisme des *vues* de SSREFLECT, permettant de jongler entre les interprétations logiques et computationelle dans le langage de preuve, mais dont nous ne détaillerons pas l'élégant usage.

De plus, pour que la relation d'équivalence du domaine sur lequel on travaille soit adéquate par rapport à cette réflection, on base le développement sur les éléments de la structure `eqType` ci-dessous, au lieu des éléments de `Type` usuellement employés.

```
Structure eqType : Type := EqType {
  sort :> Type ;
  eq : sort → sort → bool ;
  eqP : forall x y, reflect (x = y) (eq x y)
}.
End EqType.
```

Ici, `sort` est une coercion de `eqType` vers son support, `eq` est la fonction qui décide de l'égalité, et `eqP` le théorème qui injecte cette égalité dans celle de Leibniz, ce qui permet de faire de `eq` une relation réécrivable.

2.2 Structures Canoniques

Il devient donc très simple de manipuler les formulations booléennes décrivant des relations entre deux éléments d'un type inductif : il suffit pour cela d'utiliser la relation réfléchie `eq` définie dans la structure `eqType` correspondant à ce type. La structure `eqType` à employer pour un `Type` donné est inférée par le typechecker à l'aide des *structures canoniques* : déclarer un objet `eqType` comme structure canonique en fait un argument implicite par défaut pour résoudre les équations impliquant des objets dont elle est le sous-type.

La formalisation que nous avons développé se prêtait bien à l'usage de ces déclarations puisque, au niveau de précision que nous souhaitions atteindre, la relation étudiée sur les types comme les termes était l'égalité. La manipulation de ces objets sous forme booléenne, très concise, ne nous a donc coûté que la définition de structures `eqType` correspondant à ces types, dotés d'une égalité correspondant (via quelques indirections éventuelles, voir partie 7) au test d'égalité structurelle, et d'une déclaration de cette structure comme canonique.

3 General setting

3.1 Deep vs. shallow

The denomination "normalization by evaluation" is now well-established and designates a class of techniques dealing with functional programs. However, depending upon the framework these techniques are used in, the aim can vary greatly. In all cases, the idea is to translate functional programs (λ-terms) from one level of language to another. This is precisely what we do here. In type theory, a λ-term can exist under very different forms. Let us consider the term $\lambda x.\lambda y.x$. Given any type X of the type theory, we can build the

corresponding program ; in Coq syntax it will be `fun (x:X)(y:X) => x`. This is what is generally called the *shallow* embedding.

We can however also go for the *deep* embedding. This involves defining a data-type describing λ-terms. Again in Coq we can take :

```
Inductive term : Type :=
 Var : id -> term | Lam : id -> term -> term
  | App : term -> term -> term.
```

where `id` is a well-chosen datatype. The same term is then represented as the object `Lam x (Lam y (Var x))` of type `term`.

Given a λ-term, each of the representations has its advantages and shortcomings. Advantages of the deep embedding representation include :

- We "know" the code of the term, for instance we can print it, compute its length, etc.
- Correspondingly, we can reason by induction over the structure of the λ-term.
- We can define properties of λ-terms (typing, reduction...) by induction or by recursion over the term's structure.

All these things are, at first, impossible with the shallow encoding. On the other hand, there are tasks where the latter has its advantages :

- With the shallow representation, we do not "know" the code, but we can run it. Furthermore, the λ-terms are logically identified modulo β-conversion.
- With the deep encoding, one has to make the dreaded choice between de Bruijn indexes and named variables ; both coming with their own, well-known, difficulties (explicit lifting functions with de Bruijn, treatment of α-conversion with named variables). With the shallow encoding, this is basically subcontracted to the proof-system's implementation.

It is therefore an interesting question to what extent one can switch between the two representations and recover one from the other. It is this question we try to address in this work.

Terminology

Since the shallow encoding corresponds to an executable, but not readable, representation, we will speak of the *compiled representation*. On the other hand, the deep encoding's representation we will call the *decompiled* or *source level* representation.

3.2 Normalization by evaluation

The more difficult translation is going from the compiled to the source code. On the compiled level, the inner code of functions cannot be accessed : given a function, the only possible operation is to apply an argument to it, evaluate and examine the result. We can think of fuctions as being represented as closures in an abstract machine. Opening these closures and reconstructing the original code is precisely what *Normalization by Evaluation* (NbE) is about.

The result of this decompilation process is always in normal form. This is, of course, the reason for the terminology "normalization by evaluation" which was coined by Ulrich Berger. To be precise, the result of the decompilation is in *strong* normal form, which means that code under binders is also in normal form. A consequence is that NbE is used in practice for partial evaluation of functional programs.

The content of this work can thus roughly be summed up as being a formal treatment of a particular kind of NbE in Coq. Dependent type theories, like Coq, are a very interesting framework for using NbE for several reasons :

- The decompilation function takes an argument whose type will vary. When working in a conventional language like ML, one needs, for instance, to insert an additional layer (like HOAS in [DRR01]) in order to accommodate this. Using dependent types, the decompilation is naturally typed.
- Functional programs are, by essence, objects of type theory (compiled form). But in many cases, for instance when doing metatheory, they are also objects of study inside the type theory. One will then want to reason about the syntax of programs, thus using the decompiled form. NbE is precisely the way to switch from the first encoding to the second.
- Finally, since type-theories are a framework where one can, at the same time, do computations and proofs, we can not only write the NbE routines, but also specify and certify their behavior.

3.3 The general idea

We outline the framework by recalling the basic ideas in the case of a functional programming language with side-effects. We therefore use Caml syntax in this section.

We write programs performing NbE over simply typed λ-terms. The routines use type information in an essential way. We thus first need a data-type for representing simple types as well as one representing the terms :

```
type st = Iota | Arr of st*st ;;

type term = Var of string | Lam of string*term
                | App of term * term ;;
```

It is well-known that an important point is being able to produce fresh variables. In this first rough description, we "cheat" by relying on side-effects : we use a `gensym` function returning a new string each time it is called.

We then can program the `decomp` decompilation function ; as usual it is defined simultaneously with a function "compiling" values. We here call this function `long` since it builds a η-long form of variables. Both functions are known under various names in the literature (resp. reify, reflect...).

```
let rec decomp t = function
  | Iota       -> t
  | Arr(u1,u2) -> let x = gensym() in
```

```
      Lam(x,(decomp (t (long (Var(x)) u1)) u2))

and long t = function
  | Iota       -> t
  | Arr(u1,u2) -> fun x->(long (App(t,(decomp x u1))) u2);;
```

The type of the argument `t` of `decomp`, as well as the type of the result of `long`, depend on their second argument and the programs are therefore not typable in this form. However, if we locally switch off the type-checker, we can see, for instance, that the evaluation of `(decomp (Arr Iota Iota) (fun x->x))` indeed returns something of the form `Lam(`s`,Var(`s`))` where s corresponds to the current state of `gensym`.

3.4 Syntax vs. semantics point of view

Looking at what is going on through the lens of the Curry-Howard isomorphism, one obtains another interesting explanation : typed λ-terms standing for proofs, the deep encoding corresponds to the syntactical notion of provability. The compiled/shallow encoding, on the other hand, is semantical : it is a translation of the logic of simple types into the type theoretical framework. Having understood this, it appears that NbE corresponds to a *completeness result*, since it lifts results from the semantics back into the syntax. This remark is enlightening even if it is less central in the present work than in the ones of Catarina Coquand and Jean-Louis Krivine.

3.5 Related work

We are not aware of too many actually implemented formal treatments of NbE. The main one is indeed Catarina Coquand's pioneering work [Coq93] which we saw first presented at the 1992 European Types meeting ; a later version can today be found in [Coq02]. The motivation however is not exactly the same, and in consequence we can list some technical differences :

- As mentioned above, Catarina Coquand's point of view is more on the logical side and the result is understood as a completeness result. The compiled level is understood as a semantics. Thus, semantics are always defined with respect to a context ; which we will try to avoid. Also, she uses dependent types in a much more intensive way than what we do.
- The main technical difference however is the type of the semantics. She uses a Kripke-style typing of the compiled terms, which is useful for stating and proving completeness. On the other hand, it makes the actual compiled/semantical code more complex, which is what we try to avoid.

The Kripke-style typing of the semantics can also be found, among others, in the work by Altenkirch et al. [ADHS01], which extends NbE to sum types.

Even closer to us is the line of work initiated by Ulrich Berger and Helmut Schwichtenberg [Ber93, BES98, BS91, BES03]. However, and even if their constructions are precisely described, the only actual implementation is not exactly NbE, but a formalization of Tait's

strong normalization proof [BBLS06]. Ulrich Berger also showed that Tait's proof is actually what underlies NbE algorithms ; but this analogy is not explicit in the formalization. Technically, if we leave the issue of formalization aside, the main difference with Berger's NbE is due to the fact that again we use a (slightly) simpler typing of the compiled terms. We have to sacrifice the efficiency of the normalization algorithm for that, whereas efficiency was precisely his main motivation.

A lot of very detailed work has, of course, been devoted to NbE by Olivier Danvy. He does not set his work in type theory, but his ideas are obviously influential in this and the related works. Since his original motivation was partial evaluation, the same technique is called *type directed partial evaluation* in his work. Finally, as mentioned above, Jean-Louis Krivine's work [Kri96] on the completeness theorem, although very different bears also some relations with ours.

3.6 About the formal Coq development

In what follows, we indicate the Coq name of the definitions and lemmas, so one can relate the article with the formal proof[1].

4 The syntax of simply typed λ-calculus

The first data-type we need is a representation of simple types. In the present case, we only use one atomic type we will call `Iota` in reference to Church. It seems obvious however that the whole development could accommodate several atomic types without too much effort. We thus have an inductive type with two constructors ; in Coq syntax :

```
Inductive ST : Set := Iota : ST | Arr : ST -> ST -> ST.
```

From now on, except for short Coq verbatim, we write $\Rightarrow$ for `Arr` and ι for `Iota`. Thus $\iota \Rightarrow \iota \Rightarrow \iota$ stands for `(Arr Iota (Arr Iota Iota))`. When studying simply-typed λ-calculus, it is often convenient to have the type of the variable be part of its name (or identifier), because it precludes the use of context for defining typing. We thus take :
`Record id : Type := mkid {idx : nat ; idT : ST}.`
In the present description we write x^A for $(\texttt{mkid}\ x\ A)$, except in Coq verbatim.

We can now define the syntax of the language :

```
Inductive term : Type :=
 Var : id -> term | Lam : id -> term -> term
  | App : term -> term -> term.
```

We stress that the natural number component of the identifiers should really be understood as a "name" and not as anything even remotely related to de Bruijn indexes. This is

[1] The formal development is available on the web at
`http ://www.eleves.ens.fr/home/garillot/nbehoas.tgz`

reflected in the type of the LAM constructor, and in the definition of the list of free variables (representing the finite set of their identifiers), as the recursive function verifying the following equations :

$$\begin{aligned} \mathrm{FV}(\mathsf{Var}\ x) &\equiv \{x\} \\ \mathrm{FV}(\mathsf{App}\ t\ u) &\equiv \mathrm{FV}(t) \cup \mathrm{FV}(u) \\ \mathrm{FV}(\mathsf{Lam}\ x) &\equiv \mathrm{FV}(t) \setminus \{x\} \end{aligned}$$

We also define a function generating a fresh identifier, that is, given a type, an identifier of this type which does not belong to a given finite set :

$$\mathsf{fresh} : (\mathsf{set}\ \mathtt{id}) \to \mathtt{ST} \to \mathtt{id}$$
$$\mathsf{fresh}\ l\ T \equiv j^T \text{ where } j = max\{i, i^T \in l\} + 1$$

5 Decompilation

We now have to make the most important choice in this development. Namely the type of the objects on the compiled side. What differentiates our work with previous ones is that we here take, from the start, the simplest possible translation described by the following equations :

$$\begin{aligned} \mathsf{tr}(\iota) &\equiv \mathsf{term} \\ \mathsf{tr}(A \Rightarrow B) &\equiv \mathsf{tr}(A) \to \mathsf{tr}(B) \end{aligned}$$

This means that we want to define decompilation such that :

$$\begin{aligned} \mathsf{decomp} &: \forall T : \mathsf{ST}, \mathsf{tr}(T) \to \mathsf{term} \\ \mathsf{long} &: \mathsf{term} \to \forall T : \mathsf{ST}, \mathsf{tr}(T) \end{aligned}$$

Note that this means that decomp will return a term even when applied to a "pathological" object (for instance a function f : term $\to$ term which discriminates between β-convertible terms). The "well-behaved" objects, for which the decompilation result is meaningful, are characterized in section 8.

We are now back at the question of fresh variables : when decompiling a function $f : \mathsf{tr}(A) \to \mathsf{tr}(B)$, we have to "guess" what free variables are "hidden" inside it. What are the possible options ?

- Berger deals with the problem by having the program to be decompiled computing not only its own normal form, but also the variables that were used in order to decompile it. This results in a very efficient normalization function, but in the definition of tr, term has to be replaced by nat $\to$ term. The main negative consequence is that terms which originally contain free variables are more delicate to deal with.
- If we want to emulate the behavior of the imperative *gensym* used in the introduction, we run into a similar problem. We would need to perform a state-passing-style transformation over the program. Not only the decomp program, but also the program f we want to decompile ; which means again changing the definition of tr.

We here explore another possibility, in which we sacrifice efficiency to salvage the simplicity of tr : we compute the set of free variables inside f by first decompiling f after applying a "dummy" variable to it. We then can find a fresh variable and decompile it again. Our definition thus reads :

$$\begin{array}{rcl} \mathsf{decomp}\ \iota\ t & \equiv & t \\ \mathsf{decomp}\ A \Rightarrow B\ f & \equiv & \mathsf{let}\ x = \mathsf{fresh}\ (\mathsf{FV}(\mathsf{decomp}\ B\ (f\ (\mathsf{long}\ \mathsf{Var}(\mathsf{dum})\ A)))\ A \\ & \mathsf{in} & \mathsf{Lam}(x, \mathsf{decomp}\ B\ (f\ (\mathsf{long}\ x\ A))) \end{array}$$

where dum is a particular identifier chosen for this purpose.

Of course, since it uses two recursive calls instead of one, the algorithm is exponentially slower than the "reasonable" options above. The advantage is that it is easy to build terms on the semantical side. For example decompiling $\mathsf{Lam}\ x\ (Var\ y^\iota)$ will return the constant function `fun` a=>Var t^ι.

One technicality here is the issue of the freshness of the dummy itself. When given a term $f\ (\mathsf{long}\ \mathsf{Var}(\mathsf{dum})\ A)$, there is no obvious way to tell whether $\mathsf{Var}(\mathsf{dum})$ itself is indeed free in (the term corresponding to) f. The idea here is that the β-conversion relation is such that simply observing the *structure* of a β-normal object obtained from the application of a "dummy" to a term will allow us to infer the free variables of this term.

This is made precise in our development, in which the following lemma comes up as a requirement :

5.1 Lemma [`fvc`] :

$$\forall x\ t\ u,\ (\mathsf{App}\ t\ (\mathsf{Var}\ x) =_{\beta\eta} u \wedge y \notin \mathsf{FV}(u)) \Rightarrow \exists v, (t =_{\beta\eta} v \wedge y \notin \mathsf{FV}(v))$$

This lemma is essentially about untyped λ-calculus. Its proof is quite intricate when going into the details of α-conversion. The formalization involves a number of technical results about β-conversion in a named setting. We provide a formal proof, admitting two well-known properties of the untyped β-reduction : the postponability of α-conversion, and the Church-Rosser property.

6 Basic syntactic definitions

6.1 Typing

As we have seen, decompilation can be defined independently of typing, reduction and the usual basic notions about λ-terms. These definitions are however obviously necessary for going on. We describe a practical definition of typing. Typing is a binary relation between a term and a type. We can, of course use the inductive predicate corresponding to the usual rules :

$$\frac{}{n^T : T} \qquad \frac{t : T}{\mathsf{Lam}\ n^U t : U \Rightarrow T} \qquad \frac{t : U \Rightarrow T \quad u : U}{\mathsf{App}\ t\ u : T}$$

It is then possible to show that it is decidable whether a term bears a type or not. In practice, it appears convenient to proceed the other way around starting with the inference function and then using it to define well-typedness.

$$\begin{array}{rcll}
\mathsf{infer}(\mathsf{Var}(x^A)) & \equiv & \mathsf{Some}(A) & \\
\mathsf{infer}(\mathsf{Lam}(x^A,t)) & \equiv & \mathsf{None} & \text{if } \mathsf{infer}(t) = \mathsf{None} \\
\mathsf{infer}(\mathsf{Lam}(x^A,t)) & \equiv & \mathsf{Some}(A \Rightarrow B) & \text{if } \mathsf{infer}(t) = \mathsf{Some}(B) \\
\mathsf{infer}(\mathsf{App}(t,u)) & \equiv & \mathsf{Some}(B) & \text{if } \mathsf{infer}(t) = \mathsf{Some}(A \Rightarrow B) \\
 & & & \text{and } \mathsf{infer}(u) = \mathsf{Some}(A) \\
\mathsf{infer}(\mathsf{App}(t,u)) & \equiv & \mathsf{None} & \text{in the other cases}
\end{array}$$

It is then easy to define well-typedness :
– A term t is well-typed iff $\mathsf{infer}(t) \neq \mathsf{None}$
– the term t is of type T iff $\mathsf{infer}(t) = \mathsf{Some}(T)$

A technical point which is useful for the actual feasibility of the development is that since equality is (obviously) decidable over `ST`, the equalities above are *booleans* (which can when necessary be coerced to propositions). Together with Georges Gonthier's SSR proof tactic package [Gon05b], this makes many proofs shorter and easier. We come back to this in section 7.

Proving equivalence between the two formulations of typing is easy (10 lines).

6.2 Substitution

Because of the necessary renaming operations, the primitive substitution operation needs to be defined over several variables. A substitution is a list of pairs (x,t) meaning that the variables of identifier x are to be substituted by the term t. The recursive substitution function is defined by the following equations :

$$\begin{array}{rcl}
(\mathsf{App}\ t\ u)[\sigma] & \equiv & \mathsf{App}\ t[\sigma]\ u[\sigma] \\
(\mathsf{Var}\ x)[\sigma] & \equiv & \mathsf{Var}\ x \quad \text{if no } (x,u) \text{ occurs in } \sigma \\
(\mathsf{Var}\ x)[\sigma] & \equiv & u \qquad \text{if } (x,u) \text{ is the first occurrence of } x \text{ in } \sigma \\
(\mathsf{Lam}\ x\ t)[\sigma] & \equiv & \mathsf{Lam}\ x'\ t[(x, \mathsf{Var}\ x') :: \sigma] \\
 & & \text{where } x' = \mathsf{fresh}\left(\bigcup_{z \in \mathbf{FV}(t)\setminus\{x\}} FV(\mathsf{Var}\ z[\sigma])\right)
\end{array}$$

Subsitution is primitively defined as handling several variables simultaneously. This allows us to define substitution through structural recursion over the substituted term rather than over its size and considerably simplifies proofs.

Moreover, when applying a substitution, we systematically rename all bound variables in a uniform manner, using fresh, a deterministic choice function defined up to the set of variables it takes as an argument. This set is chosen such that is exactly characterizes the free variables of the substituted term, i.e. :

6.1 Lemma [`eFVS_FV`**]** : $\bigcup_{z \in \mathbf{FV}(t)\setminus\{x\}} FV(\mathsf{Var}\ z[\sigma]) = FV((\mathsf{Lam}\ x\ t)[\sigma])$

Following the method described by A. Stoughton [Sto88], this provides us with a way to normalize terms w.r.t. alpha-conversion when applying substitution. We indeed prove the following lemma :

6.2 Lemma [`alpha_norm`**]:** If $t =_\alpha u$, then $\forall \sigma, t[\sigma] = u[\sigma]$

We can then simply treat alpha-conversion as the equality on terms w.r.t. a substitution in the rest of the formalization, a simplification that was pivotal in making the treatment of alpha-convertibility manageable in this named setting.

Since the identifiers carry their type, it is straightforward to define what it means for a substitution to be well-typed. One then easily shows that such well-typed substitutions preserve typing.

6.3 Conversion and normal forms

Since we do not deal with strong normalization here, we can avoid the pain to define the oriented reduction relation. We just define conversion as an inductive equivalence relation.

6.3 Definition [`conv, normal, atomic`**]:** Conversion, written $=_{\beta\eta}$ is defined as the closure by reflexivity, symmetry, transitivity and congruence of the following clauses :

$$\begin{array}{ll}
(\beta) & \forall t\ u\ x, (\mathsf{App}\ (\mathsf{Lam}\ x\ t)\ u) =_{\beta\eta} t[x, u] \\
(\eta) & \forall t\ x, x \notin (\mathsf{FV}\ t) \to t =_{\beta\eta} \mathsf{Lam}\ x\ (\mathsf{App}\ t\ x) \\
(\alpha) & \forall x\ y\ t, y \notin (\mathsf{FV}\ t) \to \mathsf{Lam}\ x\ t =_{\beta\eta} \mathsf{Lam}\ y\ t[x, y]
\end{array}$$

The predicates NF (being normal) and AT (being atomic) are mutually inductively defined by the clauses :

$$\begin{array}{ll}
(\text{ATV}) & \forall x, (\text{AT Var}\ x) \\
(\text{ATAPP}) & \forall t\ u, (\text{AT}\ t) \to (\text{NF}\ u) \to (\text{AT}\ (\text{App}\ t\ u)) \\
(\text{NFAT}) & \forall t, (\text{AT}\ t) \to (\text{NF}\ t) \\
(\text{NFLAM}) & \forall x\ t, (\text{NF}\ t) \to (\text{NF}\ (\text{Lam}\ x\ t))
\end{array}$$

One can then define the sufficient conditions for the decompilation to return a normal (resp. well-typed) term :

6.4 Definition [sem_norm, sem_WT]: Given $T : \mathsf{ST}$, we define the predicates SNF T (semantic normal form) and SWT (semantically well-typed) both ranging over $\mathsf{tr}(T)$ by recursion over T :

$$\begin{array}{rcl}
\text{SNF}\ \iota\ t & \equiv & \text{NF}\ t \\
\text{SNF}\ A \Rightarrow B\ f & \equiv & \forall g : \mathsf{tr}(A), (\text{NF}\ A\ g) \to (\text{SNF}\ B\ (f\ g))
\end{array}$$

$$\begin{array}{rcl}
\text{SWT}\ \iota\ t & \equiv & \text{WT}\ t\ T \\
\text{SWT}\ A \Rightarrow B\ f & \equiv & \forall g : \mathsf{tr}(A), (\text{SWT}\ A\ g) \to (\text{SWT}\ B\ (f\ g))
\end{array}$$

6.5 Lemma [`decomp_norm`**]** : If SNF $T\ f$ (resp. SWT $T\ f$), then we have also NF $(\mathsf{decomp}\ T\ f)$ (resp. WT $(\mathsf{decomp}\ T\ f)\ T$).

PROOF. Separately, by induction over T. In each case, one has to prove simultaneously the dual condition :

$$\forall t\ T, \mathsf{AT}\ t \rightarrow \mathsf{SNF}\ T\ (\mathsf{long}\ T\ t)$$

$$\forall t\ T, \mathsf{WT}\ t\ T \rightarrow \mathsf{SWT}\ T\ (\mathsf{long}\ T\ t).$$

7 Compilation (semantics)

We now deal with going from an object t such that WT $t\ T$ holds to the corresponding object of type $\mathsf{tr}(T)$.

An environment $\mathcal{I}$ is a function which to any identifier x^X associates its semantics (an object of type $\mathsf{tr}(X)$) ; formally the type of $\mathcal{I}$ is :

```
Definition sem_env := forall (x : id) , tr x.(idT).
```

From there, given $\mathcal{I}$ we want to define the semantics of a term t such that WT $t\ T$ holds as an object of type $\mathsf{tr}(T)$ through the following, quite straightforward equations :

$$\begin{aligned}
[\mathsf{Var}\ x]_{\mathcal{I}} &\equiv \mathcal{I}(x) \\
[\mathsf{App}\ t\ u]_{\mathcal{I}} &\equiv [t]_{\mathcal{I}}([u]_{\mathcal{I}}) \\
[\mathsf{Lam}\ x^T\ t]_{\mathcal{I}} &\equiv \lambda\alpha : \mathsf{tr}(X).[t]_{\mathcal{I};\alpha\leftarrow<x,X>}
\end{aligned}$$

Formally however, this definition is not as easy to handle as it seems. The typing is delicate in the case of the application node because it requires $(\mathsf{App}\ t\ u)$ to be well-typed. The idea is that if WT $(\mathsf{App}\ t\ u)\ T$, we know there exists $U : \mathsf{ST}$ such that WT $u\ U$ and WT $t\ (U \Rightarrow T)$. This in turn implies that $[t]_{\mathcal{I}} : \mathsf{tr}(T) \rightarrow \mathsf{tr}(U)$ and $[u]_{\mathcal{I}} : \mathsf{tr}(U)$ which allows the construction $[t]_{\mathcal{I}}([u]_{\mathcal{I}})$.

If we go into detail however, the hypotheses are : there exists T, U and U', such that : $[t]_{\mathcal{I}} : \mathsf{tr}(U) \rightarrow \mathsf{tr}(T), [u]_{\mathcal{I}} : tr(U')$ and we have a proof that $U = U'$. One then has to use this last proof to transform[2] $[u]_{\mathcal{I}}$ into an object of type $\mathsf{tr}(U)$. But programs constructed that way are very difficult to reason about : because the inner types depend upon values (here the values U and U'), equational reasoning about these values can easily make the goal not well-typed anymore. Also, a naive way to define the compilation would be to have it depend upon the typing judgement. At least in the setting of Coq, this is not a good idea, again for the same reason.

[2]For Coq fans : using the `eq_rec` principle.

Treatment of type equality

Georges Gonthier suggested us a convenient way to treat type equality tests. When U and U' turn out to be equal, the result of the equality test will be used to map an object of some type $A(U)$ to $A(U')$. So we can build this coercion into the equality test's result. This is done by defining :

```
Inductive cast_result (a1 a2 : ST) : Type :=
  | Cast (k : forall P, P a1 -> P a2)
  | NoCast.
```

One then defines the function `cast : ST -> ST -> cast_result` which returns `NoCast` if its arguments are different and `Cast` applied to the identity if they are equal :

```
Fixpoint cast (a1 a2 : ST) {struct a2} : cast_result a1 a2 :=
  match a1 as d1, a2 as d2 return cast_result d1 d2 with
  | Iota, Iota => idcast
  | Arr b1 c1, Arr b2 c2 =>
    match cast b1 b2, cast c1 c2 with
    | Cast kb, Cast kc =>
      let ka P :=
        let Pb d := P (d ==> c1) in let Pc d := P (b2 ==> d) in
        fun x => kc Pc (kb Pb x)
      in Cast _ _ ka
    | _, _ => NoCast
    end
  | _, _ => NoCast
  end.
```

On one hand one then shows that `cast` actually implements the equality test. On the other hand however, one does not need to invoke this property in order to define a function like compilation. Indeed, we want to make the compilation function as robust as possible ; instead of asking for the argument to verify WT, the function is defined for any `term` but can return a default value when the argument has no semantics. This means the function returns a result of the following type :

```
Inductive comp_res : Type :=
| Comp : forall T:ST, (sem_env -> tr T) -> comp_res
| NoComp.
```

and the `comp` function is then best described by its actual code :

```
Fixpoint comp (t:term): comp_res :=
let F T := sem_env -> tr T in
  match t with
| Var i => Comp i.(idT) (fun I => (I i))
```

```
| Lam i t =>
   match comp t with
  | NoComp => NoComp _
  | Comp U f => Comp (Arr i.(idT) U)
                     (fun I x => (f (esc I i x)))
   end

| App u v =>
   match comp u , comp v with
  | Comp (U1 ==> U2) su , Comp V sv =>
     match cast V U1 with
       | Cast f => Comp U2 (fun I => (su I (f tr (sv I))))
       | NoCast => NoComp _
     end
  | _,_ => NoComp _
 end
end.
```

The two main remarks are :

- One easily proves that comp correctly re-implements type-checking. That is $(\texttt{comp}\ t)$ is of the form $(\texttt{Cast}\ T\ v)$ if and only if $(\mathsf{infer}\ t)$ reduces to $(\mathsf{Some}\ T)$.
- The big advantage of using `cast` appears, as expected, in the clause for `App t u` : we use the cast `f` in order to construct the result, without having to know that `cast` actually implements equality. This may look like a technical detail but is crucial for keeping the proofs reasonably small and tractable.

One can prove a first result about compilation, namely that its result verifies the SNF property.

7.1 Lemma [`comp_norm`**]** : Let $\mathcal{I}$ be such that for all x^A, we have $\mathsf{SNF}\ A\ (\mathcal{I}\ x^A)$. If $(\mathsf{comp}\ t)$ is of the form $(\mathsf{Comp}\ T\ st)$ (which is always the case if $(\mathsf{WT}\ t\ T)$ holds), then we have $(\mathsf{SNF}\ T\ (st\ \mathcal{I}))$.

7.2 Definition [`id_env`**]** : The *standard interpretation* $\mathcal{I}_0$ is defined by

$$\mathcal{I}_0(x^T) \equiv \mathsf{long}\ T\ (\mathsf{Var}\ x^T)$$

for all x^T.

The lemmas 6.5 and 7.1 then ensure :

7.3 Theorem [`norm_norm`**]** : If $(\mathsf{comp}\ t)$ is of the form $(\mathsf{Comp}\ T\ st)$, then $(\mathsf{NF}\ (\mathsf{decomp}\ (st\ \mathcal{I}_0)))$. This is always the case when $\mathsf{WT}\ t\ T$ holds.

8 The central logical relation

It then remains to show that compilation and decompilation preserve conversion. That is if WT $t\ T$ and if n and $\mathcal{I}$ are well-chosen, then

$$t =_{\beta\eta} \mathsf{decomp}(T, n, [t]_{\mathcal{I}}).$$

Exactly like in all the related work, one here has to introduce a logical relation between the syntactic and semantic levels. The definition could not be simpler :

8.1 Definition [`sem_conv`**]:** If $t : \mathsf{term}, T : \mathsf{ST}$ and $f : \mathsf{tr}(T)$ then $t \simeq_T f$ is a proposition defined recursively by :

$$\begin{array}{rcl} t \simeq_\iota st & \equiv & t =_{\beta\eta} st \\ t \simeq_{A\Rightarrow B} st & \equiv & \forall u\ su, u \simeq_A su \rightarrow (\mathsf{App}\ t\ u) \simeq_B (st\ su) \end{array}$$

The logical relation allows us to define an extentional equality on the semantical level :

8.2 Definition [`ext_eq`**]:** If $T : \mathsf{ST}$, then $=_T$ is a binary relation over $\mathsf{tr}(T)$ defined by :

$$\begin{array}{rcl} t =_\iota u & \equiv & t =_{\beta\eta} u \\ f =_{A\Rightarrow B} g & \equiv & \forall (t : \mathsf{term})(f'\ g' : \mathsf{tr}(A)), t \simeq_A f' \rightarrow f' =_A g' \rightarrow (f\ f') =_B (g\ g') \end{array}$$

Notice, in the second clause, that we require one of the arguments to be related to some term t. This can be understood as a way to prevent considering "pathological" functions, which, for instance, could depend upon the size of terms, the name of bound variables, etc... Although the formulation is not symmetrical, the definition actually is ; since one then shows that the logical relation is extentional.

8.3 Lemma [`sem_comp_ext1, sem_comp_ext2`**]:** Suppose $t \simeq_T st$. Then :

1. if $t =_{\beta\eta} t'$, then $t' \simeq_T st$,
2. if $st' =_T st$, then $t \simeq_T st'$.

9 Putting it all together

We can then finish the normalization proof by proving the main lemma of the development.

9.1 Lemma [`sem_comp`**]:** Let t be such that $(\mathsf{comp}\ t)$ is of the form $(\mathsf{Comp}\ T\ st)$, which is always the case when WT $t\ T$ holds. Let σ a substitution and $\mathcal{I}$ an interpretation be such that for any x^A which is free in t we have $\sigma(x) \simeq_A \mathcal{I}(x^A)$. Then we have :

$$t[\sigma] \simeq_T [t]_{\mathcal{I}}.$$

PROOF. By induction over the structure of t. It is the longest normalization-related proof of the development. It uses some technical results about free variables and α-conversion, that we have not detailed in this account.
The next lemma 9.2 states that the "well-behaved" functions, which can be decompiled, are exactly the ones related to some term by $\simeq$. Notice that this means that the lemma 9.1 can be understood as the real completeness result of the development : it states that the co-domain of the semantics are precisely the "well-behaved" functions.

9.2 Lemma [`sem_decomp`**]:** If $t \simeq_T st$, then $t =_{\beta\eta} (\mathsf{decomp}\ T\ (st\ \mathcal{I}_0))$.

PROOF. The proof of this lemma proceeds over a simple induction on the types, initially conducted simultaneously with the equivalent property on `long`. It is the only result relying on lemma 5.1, but presents no other difficulty.

It is then easy to conclude the weak normalization result :

9.3 Theorem: If WT $t\ T$, then $(\mathsf{comp}\ t)$ is of the form $(\mathsf{Comp}\ T\ st)$ and :

1. $t =_{\beta\eta} (\mathsf{decomp}\ T\ (st\ \mathcal{I}_0))$
2. NF $(\mathsf{decomp}\ T\ (st\ \mathcal{I}_0))$.

PROOF. It is a corollary of theorem 7.3, lemma 9.2 and the fact that well-typed terms are compilable. For lemma 9.2, one simply uses the fact that $t =_\alpha t[Id]$.

10 Application : représentation de la syntaxe abstraite d'ordre supérieur

10.1 Syntaxe abstraite d'ordre supérieur

La syntaxe abstraite d'ordre supérieur (HOAS, pour *higher order abstract syntax*,[PE88]) est une représentation de la syntaxe de langages de programmation de façon à éviter le traitement des variables du niveau objet. Par exemple, la syntaxe du lambda calcul simple peut être spécifiée de la façon suivante :

$$\begin{array}{rcl} \mathrm{tm} & ::= & \mathrm{app} : \mathrm{tm}, \mathrm{tm} \to \mathrm{tm} \\ & | & \mathrm{lam} : (\mathrm{tm} \to \mathrm{tm}) \to \mathrm{tm} \end{array}$$

Cette spécification signifie que le type tm des lambda expressions a deux constructeurs app et lam, le premier étant le un constructeur du premier ordre ordinaire, et le second un constructeur d'ordre supérieur prenant une fonction de niveau méta comme argument. L'avantage de cette syntaxe comparée à la syntaxe du premier ordre, telle qu'utilisée par exemple dans la partie 4 est que la substitution et l'affaiblissement sont obtenues directement, par héritage des propriétés du métalangage. On peut ainsi par exemple spécifier la β-réduction en disant que le terme $app(lam f, t)$ doit se réduire vers le terme $f(t)$, tandis qu'en syntaxe du premier ordre il est nécessaire de définir les variables libres et liées, et de définir la substitution sans capture afin d'écrire une telle règle de réduction.

Cette représentation a déjà eu des succès considérables dans le domaine de la preuve de programmes ([LCH07]), au point d'être considérée par certains comme la voie royale pour la preuve de théorèmes sur les langages. Toutefois, l'encodage direct de cette méthode en Coq est impossible, du fait de l'occurrence négative du type *tm* dans le constructeur *lam*de ce même type.

10.2 Exemple

Nous espérons pouvoir utiliser le présent développement pour raisonner sur des structures impliquant des lieurs de façon plus aisée, en implémentant des méthodes tirées de la syntaxe abstraite d'ordre supérieur. En effet, un langage avec lieur peut être encodé dans le λ-calcul simplement typé. Une manière de faire simple nous paraîtrait de décrire nos méta-termes comme des termes simplement typés, en forme normale, dont les seules variables libres seraient :

$$\text{APP} : \iota \to \iota \to \iota;\ \text{LAM} : (\iota \to \iota) \to \iota.$$

À notre développement est joint une tentative d'adoption de syntaxe abstraite d'ordre supérieur simple grâce à notre formalisation, à savoir la preuve de sûreté du typage d'un langage ainsi défini (`hoas_proof.v`). Notre but est simplement ici d'établir a facilité d'usage *syntaxique* de notre développement, en suivant celui de [HC]. Nous devons en particulier admettre deux axiomes, l'un d'extensionnalité (peu surprenant, et déjà rencontré par [HMS]), l'autre servant à s'abstraire abusivement de la syntaxe d'ordre supérieur dans la définition du cas d'abstraction du prédicat de typage : en syntaxe abstraite d'ordre supérieur, celui-ci aussi comporte une occurrence négative.

De possibles plongement de la syntaxe abstraite d'ordre supérieur ont déjà été envisagés [DPS97], [SDP01], et une poursuite possible de ce travail consisterait à se diriger dans cette voie.

11 Conclusion et travail à venir

Au cours de ce stage, nous avons mis en oeuvre et poursuivi une formalisation nommée permettant de prouver la correction de l'algorithme de normalisation par évaluation dans le prouveur Coq. Nous avons pour cela mis en oeuvre la stratégie de représentation de la substitution simultanée définie par Stoughton ([Sto88]) et les tactiques de réflection de Georges Gonthier, montrant que la réflection booléenne pouvait avoir son utilité dans le domaine des langages.

La formalisation que nous avons obtenu est complète modulo la propriété Church-Rösser pour la $\beta\eta$ réduction non typée, que nous avons admise. En effet, ces preuves sont ardues à poursuivre dans le cadre d'une formalisation nommée : même si ce cadre existe depuis longtemps, il a fallu attendre 2001 pour que les premières preuves du théorème de Church-Rösser pour la β-réduction dans un cadre nommé apparaissent ([Hom01]).

Nous avons aussi montré que ce traitement formel est possible lorsque l'on utilise les types simples comme interprétation, même si c'est au prix d'une certaine inefficacité. Il

semble difficile d'envisager qu'il soit possible de faire mieux sans changer l'interprétation des types de base vers une alternative comme celle de Berger et al.

Une première extension possible de ce travail serait donc de reprendre notre formalisation afin d'implémenter l'algorithme de [BES98]. Le gain en serait très intéressant dans un cadre comme celui de Coq, qui implémente une compilation «réelle» et autoriserait ainsi une normalisation efficace.

Toutefois, les travaux que nous avons effectué dans ce sens montrent qu'il sera sans doute difficile de réutiliser le développement que nous avons effectué ici, contrairement à ce que prévoyait [WG07]. En particulier, les lemmes que nous avions prouvé sur un méta-domaine fondé sur Λ devront maintenant s'exprimer sur $\widehat{\Lambda}$, d'une part. D'autre part, il semble intéressant d'étendre le développement de [BES98], qui s'exprime sur les termes clos, afin d'inclure tous les termes du langage objet, chose que nous avons pris le soin de faire dans notre développement (voir en particulier la partie 7). Cette extension semblerait aisée en utilisant la convention de Barendregt, et en implémentant un style de développement *locally nameless* (voir [ACP+07] et les références qui s'y trouvent). Par contre, l'implémentation de la substitution à l'aide d'une substitution simultanée, comme nous l'avons fait en 6.2 fait alors peu de sens.

Enfin, nous espérons qu'en caractérisant précisément quels sont les méta-termes qui appartiennent au fragment simplement typé du λ-calcul, et en accédant à leur syntaxe, une extension du travail effectué pourrait être une première étape pour l'implémentation de techniques inspirées par la syntaxe abstraite d'ordre supérieure dans la théorie des types, éventuellement d'une manière similaire à [CF07].

Remerciements

Outre les personnes déjà remerciées dans [WG07], l'auteur tient à remercier chaleureusement Benjamin Werner pour son encadrement, et Georges Gonthier pour ses remarques sur l'utilisation des tactiques de réflection.

Références

[ABDM03] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine : a functional derivation. Available on http ://www.brics.dk/RS/03/14/index.html, 2003.

[ACP+07] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory, July 2007. Submitted for publication.

[ADHS01] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS*, pages 303–310, 2001.

[AJ04] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalisation by evaluation. *Mathematical Structures in Computer Science*, 14(4) :587–611, 2004.

[BBLS06] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1) :25–49, 2006.

[Ber93] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 1993.

[BES98] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalisation by evaluation. In Bernhard Möller and J. V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*, pages 117–137. Springer, 1998.

[BES03] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Term rewriting for normalization by evaluation. *Inf. Comput.*, 183(1) :19–42, 2003.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *LICS*, pages 203–211. IEEE Computer Society, 1991.

[CF07] Venanzio Capretta and Amy Felty. Combining de bruijn indices and higher-order abstract syntax in coq. In Thorsten Altenkirch and Conor McBride, editors, *Proceedings of TYPES 2006*, LNCS. Springer, 2007.

[Coq93] Catarina Coquand. From semantics to rules : A machine assisted analysis. In Egon Börger, Yuri Gurevich, and Karl Meinke, editors, *CSL*, volume 832 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 1993.

[Coq02] Catarina Coquand. A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. *Higher-Order and Symbolic Computation*, 15(1) :57–90, 2002.

[Dan96] Olivier Danvy. Type-directed partial evaluation. In *POPL*, pages 242–257, 1996.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In Philippe de Groote, editor, *TLCA*, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 1997.

[DRR01] Olivier Danvy, Morten Rhiger, and Kristoffer Høgsbro Rose. Normalization by evaluation with typed abstract syntax. *J. Funct. Program.*, 11(6) :673–680, 2001.

[Fil99] Andrzej Filinski. A semantic account of type-directed partial evaluation. In *Principles and Practice of Declarative Programming*, pages 378–395, 1999.

[Fil01] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In *TLCA*, pages 151–165, 2001.

[GMR+07] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalization of finite group theory. to appear : TPHOLs 2007, Available on http ://hal.inria.fr/inria-00139131/fr/, 2007.

[Gon05a] Georges Gonthier. A computer-checked proof of the four colour theorem. Available on `http://research.microsoft.com/~gonthier/`, 2005.

[Gon05b] Georges Gonthier. Notations of the four colour theorem proof. Available on `http://research.microsoft.com/~gonthier/`, 2005.

[GP02] Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5) :341–363, 2002.

[HC] Robert Harper and Karl Crary. How to believe a twelf proof. Available on http ://www.cs.cmu.edu/ rwh/papers/how/believe-twelf.pdf.

[HMS] Furio Honsell, Marino Miculan, and Ivan Scagnetto. pi-calculus in (co)inductive type theory.

[Hom01] P. Homeier. A proof of the church-rosser theorem for the lambda calculus in higher order logic, 2001.

[Kri96] Jean-Louis Krivine. Une preuve formelle et intuitionniste du théorème de complétude de la logique classique. *Bulletin of Symbolic Logic*, 2(4) :405–421, 1996.

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 173–184. ACM, 2007.

[Mog99] Torben Æ. Mogensen. Gödelization in the untyped lambda-calculus. In *PEPM*, pages 19–24, 1999.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI*, pages 199–208, 1988.

[Pot07] François Pottier. Static name control for freshml. In *LICS*, pages 356–365. IEEE Computer Society, 2007.

[SDP01] Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.*, 266(1-2) :1–57, 2001.

[SPG03] Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. Freshml : programming with binders made simple. In Colin Runciman and Olin Shivers, editors, *ICFP*, pages 263–274. ACM, 2003.

[Sto88] Allen Stoughton. Substitution revisited. *Theor. Comput. Sci.*, 59 :317–325, 1988.

[UBN07] Christian Urban, Stefan Berghofer, and Julien Narboux. Nominal datatype package for isabelle/hol. Available from http ://wisabelle.in.tum.de/nominal/, 2007.

[Ves01] René Vestergaard. The simple type theory of normalization by evaluation. *Electr. Notes Theor. Comput. Sci.*, 57, 2001.

[WG07] Benjamin Werner and François Garillot. Simple types in type theory : deep and shallow encodings. Available at http ://www.lix.polytechnique.fr/Labo/Benjamin.Werner/nbe.html, 2007.