Simple Types in Type Theory: deep and shallow Encodings

François Garillot[1] and Benjamin Werner[2*]

[1] Ecole Normale Supérieure
45 rue d'Ulm, Paris, France
and INRIA-Futurs at
LIX, Ecole Polytechnique, Palaiseau, France
francois.garillot@ens.fr
[2] INRIA-Futurs at
LIX, Ecole Polytechnique, Palaiseau, France
Benjamin.Werner@inria.fr

Abstract. We present a formal treatment of normalization by evaluation in type theory. The involved semantics of simply-typed λ-calculus is exactly the simply typed fragment of the type theory. This means we have constructed and proved correct a decompilation function which recovers the syntax of a program, provided it belongs to the simply typed fragment. The development runs and is checked in Coq. Possible applications include the formal treatment of languages with binders.

1 General setting

1.1 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.
```

* This work was partly supported by the The Microsoft Research-INRIA Joint Centre.

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.

1.2 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 [10]) 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.

1.3 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`.

1.4 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.

1.5 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 [8] which we saw first presented at the 1992 European Types meeting; a later version can today be found in [9]. 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. [1], which extends NbE the sum types.

Even closer to us is the line of work initiated by Ulrich Berger and Helmut Schwichtenberg [2, 4, 6, 5]. 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 [3]. 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 [14] on the completeness theorem, although very different bears also some relations with ours.

1.6 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[3].

2 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 (`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 reflected in the type of the `Lam` constructor, and in the definition

[3] The formal development is available on the web at
`http://www.lix.polytechnique.fr/Labo/Benjamin.Werner/`

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:

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

3 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 $f : \mathsf{term} \to \mathsf{term}$ which discriminates between β-convertible terms). The "well-behaved" objects, for which the decompilation result is meaningful, are characterized in section 6.

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 → 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:

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

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 y^{ι}.

One technicality here is the issue of the freshness of the dummy itself. When given a term f (long Var(dum) A), there is no obvious way to tell whether Var(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:

Lemma 1 (`fvc`).

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

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.

4 Basic syntactic definitions

4.1 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.

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

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 [13], this makes many proofs shorter and easier. We come back to this in section 5.

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

4.2 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:

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

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.:

Lemma 2 (`eFVS_FV`). $\bigcup_{z \in FV(t) \setminus \{x\}} FV(\mathsf{Var}\ z[\sigma]) = FV(t[\sigma])$

Following the method described by A. Stoughton [17], this provides us with a way to normalize terms w.r.t. alpha-conversion when applying substitution. We indeed prove the following lemma:

Lemma 3 (`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.

4.3 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.

Definition 1 (`conv, normal, atomic`). *Conversion, written* $=_{\beta\eta}$ *is defined as the closure by reflexivity, symmetry, transitivity and congruence of the following clauses:*

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

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

$$
\begin{array}{ll}
(\textsc{AtV}) & \forall x, (\mathrm{AT}\ \mathsf{Var}\ x) \\
(\textsc{AtApp}) & \forall t\ u, (\mathrm{AT}\ t) \to (\mathrm{NF}\ u) \to (\mathrm{AT}\ (\mathsf{App}\ t\ u)) \\
(\mathrm{NFAt}) & \forall t, (\mathrm{AT}\ t) \to (\mathrm{NF}\ t) \\
(\mathrm{NFLam}) & \forall x\ t, (\mathrm{NF}\ t) \to (\mathrm{NF}\ (\mathsf{Lam}\ x\ t))
\end{array}
$$

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

Definition 2 (**sem_norm, sem_WT**). *Given* $T : \mathsf{ST}$, *we define the predicates* $\mathrm{SNF}\ T$ *(semantic normal form) and* SWT *(semantically well-typed) both ranging over* $\mathsf{tr}(T)$ *by recursion over* T*:*

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

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

Lemma 4 (`decomp_norm`). *If* SNF T f *(resp.* SWT T f*), then we have also* NF (decomp T f) *(resp.* WT (decomp T f) T*).*

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

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

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

5 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 (App t u) to be well-typed. The idea is that if WT (App t u) T, we know there exists U : ST such that WT u U and WT t ($U \Rightarrow T$). This in turn implies that $[t]_{\mathcal{I}} : \mathsf{tr}(T) \to \mathsf{tr}(U)$ 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: $[t]_{\mathcal{I}} : \mathsf{tr}(U) \to \mathsf{tr}(T), [u]_{\mathcal{I}} : tr(U')$ and we have a proof that $U = U'$. One then has to use this last proof to transform[4] $[u]_{\mathcal{I}}$ 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.

[4] 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 (`comp` t) is of the form (`Cast` T v) if and only if (infer t) reduces to (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.

Lemma 5 (`comp_norm`). *Let $\mathcal{I}$ be such that for all x^A, we have* SNF A ($\mathcal{I}$ x^A). *If* (comp t) *is of the form* (Comp T st) *(which is always the case if* (WT t T) *holds), then we have* (SNF T (st $\mathcal{I}$)).

Definition 3 (`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 4 and 5 then ensure:

Theorem 1 (`norm_norm`). *If* (comp t) *is of the form* (Comp T st), *then* (NF (decomp (st $\mathcal{I}_0$))). *This is always the case when* WT t T *holds.*

6 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:

Definition 4 (`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:*

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

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

Definition 5 (`ext_eq`). *If* $T : \mathsf{ST}$, *then* $=_T$ *is a binary relation over* $\mathsf{tr}(T)$ *defined by:*

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

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.

Lemma 6 (`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* $t \simeq_T st'$.

7 Putting it all together

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

Lemma 7 (`sem_comp`). *Let* t *be such that* $(\mathsf{comp}\ t)$ *is of the form* $(\mathsf{Comp}\ T\ st)$, *which is always the case when* $\mathrm{WT}\ t\ T$ *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 8 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 7 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.

Lemma 8 (`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 1, but presents no other difficulty.

It is then easy to conclude the weak normalization result:

Theorem 2. *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 1, lemma 8 and the fact that well-typed terms are compilable. For lemma 8, one simply uses the fact that $t =_\alpha t[Id]$.

8 Conclusion and future work

We have shown that formal treatment of NbE in provers based on type theory is tractable even when using a formalization style which is very different from the one followed by Catarina Coquand. We have also shown that NbE is possible when using the simple types for the compiled terms, even if, in our case, this comes at the expense of efficiency. Whether one can get rid of this overhead without changing the typing seems however doubtful to us.

A first possible direction of work is to apply similar techniques, and possibly reuse part of this development in order to formalize and implement the related versions of NbE of Berger and his collaborators. This seems at the same time feasible and useful, especially in a framework like Coq, which implements "real" compilation and would thus allow really fast normalization.

A second more prospective but, we hope, promising direction of research is using the present development for making reasoning about structures with binders easier. Indeed, a language with binders can be described by a context of simply typed λ-calculus. For instance, untyped λ-terms can be described as simply typed λ-terms whose free variables are either of the atomic type ι, or equal to one of the two variables:

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

This is the idea of higher-order abstract syntax (HOAS [15]) whose possible use inside type theory has already been investigated [11, 16]. We hope that by allowing to characterize precisely what are the terms of the type theory which belong to the simply typed fragment and accessing their syntax, our work can be a first step for implementing techniques inspired by HOAS inside existing type theories; possibly in a way similar to [7].

Acknowledgments

We thank Thorsten Altenkirch, Thierry Coquand, Olivier Danvy, Hugo Herbelin, Marino Miculan and Helmut Schwichtenberg for helpful discussions on the topic. Anonymous referees made many useful comments. Special thanks go to Ulrich Berger who pointed out a crucial mistake and Georges Gonthier who provided the useful SSR proof tactic package and suggested the definition of typing as an actual function.

References

1. T. Altenkirch, P. Dybjer, M. Hofmann, and P. J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS*, pages 303–310, 2001.
2. U. Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 1993.
3. U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.
4. U. Berger, M. Eberl, and H. Schwichtenberg. Normalisation by evaluation. In B. Möller and J. V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*, pages 117–137. Springer, 1998.
5. U. Berger, M. Eberl, and H. Schwichtenberg. Term rewriting for normalization by evaluation. *Inf. Comput.*, 183(1):19–42, 2003.
6. U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *LICS*, pages 203–211. IEEE Computer Society, 1991.
7. V. Capretta and A. Felty. Combining de bruijn indices and higher-order abstract syntax in coq. In T. Altenkirch and C. McBride, editors, *Proceedings of TYPES 2006*, LNCS. Springer, 2007.
8. C. Coquand. From semantics to rules: A machine assisted analysis. In E. Börger, Y. Gurevich, and K. Meinke, editors, *CSL*, volume 832 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 1993.
9. C. 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.
10. O. Danvy, M. Rhiger, and K. H. Rose. Normalization by evaluation with typed abstract syntax. *J. Funct. Program.*, 11(6):673–680, 2001.
11. J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In P. de Groote, editor, *TLCA*, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 1997.
12. G. Gonthier. A computer-checked proof of the four colour theorem. Available on `http://research.microsoft.com/~gonthier/`, 2005.
13. G. Gonthier. Notations of the four colour theorem proof. Available on `http://research.microsoft.com/~gonthier/`, 2005.
14. J.-L. Krivine. Une preuve formelle et intuitionniste du théorème de complétude de la logique classique. *Bulletin of Symbolic Logic*, 2(4):405–421, 1996.
15. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI*, pages 199–208, 1988.
16. C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.*, 266(1-2):1–57, 2001.
17. A. Stoughton. Substitution revisited. *Theor. Comput. Sci.*, 59:317–325, 1988.