Abstract

In this paper I present a partial formalisation of a normaliser for type theory in Agda [Nor07]; extending previous work on big-step normalisation [AC08, AC06]. The normaliser is written as an environment machine. Only the computational behaviour of the normaliser is presented omitting details of termination.

Keywords: big-step normalisation, typed syntax, type theory, Agda, explicit substitutions

1 Introduction

This work is a small step towards the long-term goal of the author and many others to internalise the syntax [Pol95] and semantics [Dyb95, BD08] of recent formulations of type theory (with, for example, $\beta\eta$-definitional equality) and to write a type checker for type theory in type theory [BW]. The meta-theory and design of type theories is still a very active area 35 years after Martin-Löf’s early versions. Type theory is described by Martin-Löf as being intended as a “full-scale system for constructive mathematics” [ML98]. If this is to be taken seriously then it certainly should cope with the mathematics with which it is most intimately connected: its own metatheory. Internalising type theory puts great pressure on many aspects of type theory: universe hierarchy’s; inductive-recursive definitions and equality. It is not just the theory which is stretched either implementations are pushed to their limits. I believe that internalising type theory’s own notions makes up an unavoidable part of the continued development and refinement of type theory itself and is a great test for its implementations. Type theory should eat itself.

We build on previous work on big-step normalisation [AC08, AC06] and extend it to dependent types. The approach is as follows: 1. Define the well-typed terms of the language; 2. Give a simple (but not obviously terminating) normaliser as an environment machine; 3. Encode the normaliser as a big-step semantics and

---

1 Email: jmc@cs.nott.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
prove termination; 4. Augment the normaliser with the termination proof as an extra argument to yield a structurally recursive normaliser representable as a type theoretic function. For the purposes of this paper we stop short of this and focus only on the simple normaliser formalised in Agda corresponding to steps 1 and 2.

1.1 Related Work

Previous work on internal type theory\cite{Dyb95,BD08} has concerned internalising intuitionistic model theory and proofs of normalisation by evaluation (NBE)\cite{BS91}. Recently Danielsson\cite{Dan06} gave an implementation of a normaliser for the Martin-Löf Logical Framework in Agda-Light (a precursor to the version of Agda used here). The normaliser is based on NBE. It is a considerable and impressive development but it contains a number of loose ends. Firstly the soundness property of the normaliser is not shown. Secondly the development uses features of uncertain foundation. The inductive definition of semantic values is not strictly positive and the there are uses of induction-recursion\cite{DS01} which do not follow the pattern of an inductive type given together with a function defined on its constructors.

Our syntax is in the style of categories with families\cite{Dyb95,Hof97} and Martin-Löf’s substitution calculus\cite{ML92}. We diverge from the usual presentation of internal categories with families\cite{Dyb95} mainly in that we include equality of contexts and our type, term and substitution equalities are heterogeneous with respect to their indices. Eg. We equate types in potentially different contexts. This, in part, leads to the inclusion of context equality. We also postulate injectivity of Π-types in the syntax.

Pollack formalised the syntax of type theory in his thesis\cite{Pol95} and proved a number of properties but not normalisation. Barras and Werner have formalised the Calculus of Constructions in Coq\cite{BW} and proved normalisation with the aim of extending this to the full theory of Coq and providing a certified kernel. Another close relative to this work is Typed Operational Semantics\cite{Gog94}.

1.2 Agda and notation

Agda is a dependently typed programming language. It supports inductive families\cite{Dyb91} and dependent pattern matching\cite{Coq92}. It has an external checker for termination. In this development we turn off the Agda’s termination checker and write programs which are not structurally recursive with the intention of showing termination later. It is not that Agda supports non-terminating functions or that it implements a type theory that does. We are just allowed the benefit of the doubt.

We present our development in Agda syntax. For the most part it looks very similar to Haskell. Infix (and mixfix) constructors are declared with underscores either side \texttt{+_+}_ and when used must be surrounded by space \texttt{m + n}. Implicit arguments (arguments which can be inferred from use) are given in braces \texttt{{a : A}}. When they cannot be inferred we supply them in braces \texttt{f {B} a} in the appropriate place or by name \texttt{f \{B = B\} a}. The keyword \texttt{forall} allows us to omit the type of an argument when quantifying when it can be inferred from use. Agda allows unicode names. We also take some extra liberties not currently supported by Agda but only for the purposes of this presentation. The complete Agda code is available online\cite{Cha07}.
use implicit quantification to omit implicit arguments in definitions of constructors. An example of this is the identity substitution which we write as \( \text{id} : \text{Sub} \Gamma \Gamma \) when in current syntax we would have to write \( \text{id} : \forall \{ \Gamma \} \rightarrow \text{Sub} \Gamma \Gamma \).

Agda supports overloading of data constructor names we take this a step further overloading type constructor and function names.

1.3 Plan of the rest of the paper

In section 2 we introduce syntax for the Martin-Löf Logical Framework. See Hofmann’s excellent tutorial article[Hof97] for a detailed description of the Martin-Löf Logical Framework and other type theories. The syntax is for the well-typed terms so is also the axiomatic semantics for the system. Sections 3 and 4 cover the normaliser. In section 3 we give a definition of weak head normal forms (which we call values) and an evaluator from the syntax to values. Section 4 covers the definition of \( \beta\eta \)-normal forms and gives a typed-directed quote function which takes values and gives normal forms. In section 5 we extend the system by providing a code for \( \Pi \)-types for the universe. Section 6 concludes the paper.

2 Syntax with Explicit Substitutions

The system presented here is the Martin-Löf Logical Framework with explicit substitutions. The judgments are as defined (mutually) as follows:

\[
\begin{align*}
\text{Con} & : \text{Set} \quad \quad \quad \quad \quad \quad \text{-- Context} \\
\text{Ty} & : \text{Con} \rightarrow \text{Set} \quad \quad \quad \quad \quad \quad \text{-- Type} \\
\text{Tm} & : \forall \Gamma \rightarrow \text{Ty} \Gamma \rightarrow \text{Set} \quad \quad \quad \quad \text{-- Term} \\
\text{Sub} & : \text{Con} \rightarrow \text{Con} \rightarrow \text{Set} \quad \quad \quad \quad \text{-- Substitution}
\end{align*}
\]

Notice that types are given together with (indexed by) their contexts and terms are given together with (indexed by) their type and context. We have no definitions for raw syntax. In type theory terms always have a type and by giving the constructors of a type we explain what it means to be a term of that type.

Also for type theory (where the Martin-Löf LF is one such example), as we shall see for \( \text{coe} \), we cannot separate the equation syntax from the syntax of well-typed terms. Hence the corresponding equality judgments must also be mutually defined:

\[
\begin{align*}
\text{=} & : \text{Con} \rightarrow \text{Con} \rightarrow \text{Set} \quad \quad \quad \quad \text{-- Equality of contexts} \\
\text{=} & : \text{Ty} \Gamma \rightarrow \text{Ty} \Gamma' \rightarrow \text{Set} \quad \quad \quad \quad \text{-- Equality of types} \\
\text{=} & : \text{Tm} \Gamma \sigma \rightarrow \text{Tm} \Gamma' \sigma' \rightarrow \text{Set} \quad \quad \quad \quad \text{-- Equality of terms} \\
\text{=} & : \text{Sub} \Gamma \Delta \rightarrow \text{Sub} \Gamma' \Delta' \rightarrow \text{Set} \quad \quad \quad \quad \text{-- Equality of substitutions}
\end{align*}
\]

Notice that this is the decidable definitional equality that a typechecker would use. We do not consider propositional equality here. By including the definitional equality and types and contexts in the syntax we are in effect encoding typing derivations as part of the syntax.

We now explain one at a time how to construct elements of these eight sets. Starting with contexts. Contexts are left-to-right sequences of types. We use de Bruijn indices[dB72] so the contexts do not carry names. There are two ways to construct a context; either it is the empty context or it is an existing context given
together with a type from that context.

**mutual**

```plaintext
data Con : Set where
    ε : Con
    _·_ : forall Γ → Ty Γ → Con
```

Types are indexed by contexts:

```plaintext
data Ty : Con → Set where
```

The syntax is fully explicit about uses of the type (and context) coercions. These correspond to the conversion rules in the traditional syntax. There is a constructor for this even at the level of types. Given a type in context \( \Gamma \) and \( \Gamma \) and \( \Delta \) are equal contexts then we have a type in \( \Delta \).

```plaintext
coe : Ty \( \Gamma \rightarrow \Gamma = \Delta \rightarrow Ty \Delta
```

There is a constructor for explicit substitutions.

```plaintext
_[_]: Ty \( \Delta \rightarrow Sub \Gamma \Delta \rightarrow Ty \Gamma
```

In Danielsson’s presentation of the syntax of this system he does not include explicit substitutions at the level of types instead choosing to define a type-level substitution operation mutually with the syntax. His approach has the advantage of simplifying the treatment of semantic equality and reduces the number of properties that must be postulated in the syntax. However this cannot readily be extended to more complex systems such as having a universe closed under \( \Pi \)-types which we consider in section 5. The remaining constructors cover the universe (which contains only neutral terms in the Logical Framework), embedding codes for types from the universe into types and dependent functions (\( \Pi \)-types). Notice that the second argument (the range) to \( \Pi \) contains has a extra variable in its context (the domain).

```plaintext
U : Ty \( \Gamma
El : Tm \( \Gamma \rightarrow Ty \Gamma
\Pi : (\sigma : Ty \Gamma) \rightarrow Ty(\Gamma, \sigma) \rightarrow Ty \Gamma
```

Terms are indexed by context and type and include explicit constructors for conversion and substitutions:

```plaintext
data Tm : forall \( \Gamma \rightarrow Ty \Gamma \rightarrow Set where
    coe : Tm \( \Gamma \sigma \rightarrow \sigma = \sigma' \rightarrow Tm \Gamma \sigma'\)
    _[_]: Tm \( \Delta \sigma \rightarrow (ts : Sub \Delta) \rightarrow Tm \Gamma (\sigma[ts])
```

Variables (de Bruijn indices[dB72]) are not singled out as a separate syntactic class in the syntax. Instead we have \( \text{top} \) which is the first bound variable and then other variables are obtained by applications of the weakening substitution (which is called \( \text{pop} \) and is introduced ed below) (\( \text{top}[\text{pop} \sigma], \text{top}[\text{pop} \sigma \cdot \text{pop} \tau \], etc.) to \( \text{top} \). This treatment of variables is present in categorical treatments of type theory such as categories with families[Hof97,Dyb91]

```plaintext
\text{top} : Tm (\Gamma, \sigma)(\sigma[\text{pop} \sigma])
```

The categorical combinator for application is included rather than conventional application as it simplifies the presentation of the syntax and evaluation. The
standard notion of application is defined later as a convenience.

\[
\begin{align*}
\lambda & : \text{Tm} (\Gamma, \sigma) \tau \rightarrow \text{Tm} (\Pi \sigma \tau) \\
ap & : \text{Tm} (\Pi \sigma \tau) \rightarrow \text{Tm} (\Gamma, \sigma) \tau
\end{align*}
\]

Next are substitutions. Instead of presenting substitutions as just sequences of terms and then defining identity and composition recursively we give constructors for all the necessary operations. This allows the syntax to remain first-order.

\[
data \text{Sub} : \text{Con} \rightarrow \text{Con} \rightarrow \text{Set} \text{ where} \\
\text{coe} : \text{Sub} \Gamma \Delta \rightarrow \Gamma = \Gamma' \rightarrow \Delta = \Delta' \rightarrow \text{Sub} \Gamma' \Delta' \\
\text{id} : \text{Sub} \Gamma \Gamma \\
\text{pop} : \forall \sigma \rightarrow \text{Sub} (\Gamma, \sigma) \Gamma \\
\text{_<-} : (\text{ts} : \text{Sub} \Gamma \Delta) \rightarrow \text{Tm} \Gamma (\sigma [\text{ ts }]) \rightarrow \text{Sub} \Gamma (\Delta , \sigma) \\
\text{_<-} : \text{Sub} \Gamma \Delta \rightarrow \text{Sub} \beta \Gamma \rightarrow \text{Sub} \beta \Delta
\]

Some smart constructors are introduced as a notational convenience. These are just simple non-recursive functions which compute to common uses of the constructors. This actually makes our syntax inductive recursive but the smart constructors could just be expanded to avoid this. We give only their type signatures here. The first two are one place substitution on types and terms respectively. The next allows us to apply \(\text{El}\) to a term whose type is the constant \(\mathsf{U}\) applied to a substitution without using a coercion directly. Whilst this type is equal to \(\mathsf{U}\) according to the equality judgement of our system it is not definitionally equal in the metatheory. Next is conventional application (and a variation including substitutions) which is useful for embedding neutral applications back into terms. Finally we have a weakening substitution which allows us to push substitutions under binders more easily than using \(\text{pop}\) and \(\text{top}\) directly.

\[
\begin{align*}
\text{sub}^* & : \text{Ty} (\Gamma, \sigma) \rightarrow \text{Tm} \Gamma \sigma \rightarrow \text{Ty} \Gamma \\
\text{sub} & : \text{Tm} (\Gamma, \sigma) \tau \rightarrow (a : \text{Tm} \Gamma \sigma) \rightarrow \text{Tm} \Gamma (\text{sub}^* \tau a) \\
\text{El}^* & : \{\text{ts} : \text{Sub} \Gamma \Delta\} \rightarrow \text{Tm} \Gamma (\mathsf{U} [\text{ ts }]) \rightarrow \text{Ty} \Gamma \\
\text{-$_<$} & : \text{Sub} \Gamma \Delta \rightarrow \{a : \text{Tm} \Gamma \sigma\} \rightarrow \text{Tm} \Gamma (\text{sub}^* \tau a) \\
\text{-$_<$} & : \{\text{ts} : \text{Sub} \Gamma \Delta\} \rightarrow \text{Tm} \Gamma (\Pi \sigma \tau [\text{ ts }]) \rightarrow \\
& \hspace{1cm} (a : \text{Tm} \Gamma (\sigma [\text{ ts }])) \rightarrow \text{Tm} \Gamma (\tau [\text{ ts < a }]) \\
\text{-$_<$} & : \{\text{ts} : \text{Sub} \Gamma \Delta\}(\sigma : \text{Ty} \Delta) \rightarrow \text{Sub} (\Gamma, \sigma [\text{ ts }]) (\Delta, \sigma)
\end{align*}
\]

Having described the first four judgment forms we go on to consider their corresponding equality judgments which are defined mutually as dictated by the coercions. The definitions are quite long and we omit various details here. Many of the rules that make up the equality relations might be described as boilerplate and this has a variety of sources. Firstly there are rules for equivalence and congruences for data constructors. Then there are rules induced by the explicit substitutions interacting with the data constructors. The coercions induce coherence conditions on types, terms, and substitutions similar to those present in heterogeneous families of setoids. The remaining rules might be called computation rules and those are what we focus on.

We omit the definition of context equality altogether. It is just the least congruence on the type \(\text{Con}\).

\[
data \text{ _= } : \text{Con} \rightarrow \text{Con} \rightarrow \text{Set} \text{ where}
\]

5
Context equality is not always included in presentations of type theory and of its models: Hofmann\cite{Hof97} and Streicher\cite{Str91} include it; Dybjer does not\cite{Dyb95}; and Martin-Löf omits it from his presentations of type theory\cite{ML84,ML98} but includes it in the substitution calculus\cite{ML92}.

Type equality (and equality for terms and substitutions) can be represented homogeneously or heterogeneously in the sense of whether we equate types in the same context or different (but equal) contexts. Here it is presented heterogeneously as it reduces some of the bureaucracy of dealing with coercions (for example we get only one coherence condition) and the antisymmetry of the homogeneous version makes it more difficult to write the normaliser. The notion of heterogeneous equality is based on McBride’s treatment of propositional equality\cite{McB99}.

```haskell
data _=_ : Ty \Gamma \rightarrow Ty \Gamma' \rightarrow Set where
  coh : (\sigma : Ty \Gamma)(p : \Gamma = \Delta) \rightarrow coe \sigma p = \sigma
```

We skip congruence rules and equivalence rules for type equality. Next are rules that ensure that types interact appropriately with substitutions:

```haskell
rid : \{\sigma : Ty \Gamma\} \rightarrow \sigma [ id ] = \sigma
assoc : \{ts : Sub \Gamma \Delta\}{us : Sub \Gamma \Delta\} \rightarrow
  \sigma [ ts ] [ us ] = \sigma [ ts \bullet us ]
U[] : \{ts : Sub \Gamma \Delta\} \rightarrow U [ ts ] = U \Gamma
El[] : \{t : Tm \Gamma \Delta\}{ts : Sub \Gamma \Delta\} \rightarrow
  El t [ ts ] = El s (t [ ts ])
Pi[] : \{ts : Sub \Gamma \Delta\} \rightarrow
  \Pi \sigma \tau [ ts ] = \Pi (\sigma [ ts ]) (\tau [ ts / \sigma ])
```

Semantic application requires projection from equations between \Pi-types so the following constructors are added to the definitional equality. Danielsson’s simpler treatment of type equality (no explicit substitutions at the level of types) avoids this issue but it introduces induction-recursion into the definition of the syntax.

```haskell
  dom : \{\tau : Ty (\Gamma , \sigma)\}{\tau' : Ty (\Gamma' , \sigma')} \rightarrow
  \Pi \sigma \tau = \Pi \sigma' \tau' \rightarrow \sigma = \sigma'
  cod : \{\tau : Ty (\Gamma , \sigma)\}{\tau' : Ty (\Gamma' , \sigma')} \rightarrow
  \Pi \sigma \tau = \Pi \sigma' \tau' \rightarrow \tau = \tau'
```

The term equality proceeds analogously to the type equality with rules for coherence, congruence, equivalence and substitutions which are omitted. The remaining rules are the computation rules: \beta, \eta and projection from a substitution:

```haskell
data _=_ : Tm \Gamma \sigma \rightarrow Tm \Gamma' \sigma' \rightarrow Set where
  \beta : \{t : Tm (\Gamma , \sigma)\} \rightarrow ap (\lambda t) = t
  \eta : \{f : Tm \Gamma (\Pi \sigma \tau)\} \rightarrow \lambda (ap f) = f
  top< : \{ts : Sub \Gamma \Delta\}{t : Tm \Gamma (\sigma [ ts ])\} \rightarrow top [ ts < t ] = t
```

Omitting the same sets of rules for substitutions leaves the following rules:

```haskell
data _=_ : Sub \Gamma \Delta \rightarrow Sub \Gamma' \Delta' \rightarrow Set where
  lid : \{ts : Sub \Gamma \Delta\} \rightarrow id \bullet ts = ts
```

6
When defining each equality relation equations between the indices could have been included. This can be avoided by defining the following (forgetful) operations which recover these equations:

mutual
\[
\begin{align*}
\text{fog} &: \{\sigma : Ty \Gamma\} \{\sigma' : Ty \Gamma'\} \rightarrow \sigma = \sigma' \rightarrow \Gamma = \Gamma' \\
\text{fog} &: \{t : Tm \Gamma \sigma\} \{t' : Tm \Gamma' \sigma'\} \rightarrow t = t' \rightarrow \sigma = \sigma' \\
\text{fog} &: \{ts : Sub \Gamma \Delta\} \{ts' : Sub \Gamma' \Delta'\} \rightarrow ts = ts' \rightarrow \Gamma = \Gamma'
\end{align*}
\]

### 3 Values and Partial Evaluation

Values (weak-head normal forms or canonical objects) are indexed by syntactic types and contexts. The definition is very similar to our simply-typed version[AC08]. It seems possible to index values by value contexts and value types this leads to a very heavily inductive-recursive definition of values where value contexts, value types, values, the partial evaluator itself and various necessary properties must be mutually defined. This approach simplifies some of the inevitable equational reasoning imposed by the coercions but it is not clear if this is an advantage when compared with the extra complexity of the definition.

The definition presented here only requires induction-recursion to provide embeddings from values to syntax. Inevitably values must appear in types due to type dependency. On the other hand values are a subset of terms and given tool support to express this or, perhaps, just a different formulation we might not need the mutually defined embeddings.

First variables are defined and then values, neutral term and environments are defined mutually accompanied and by their respective embeddings.

**data Var :** forall \(G\) \(\rightarrow Ty G\) \(\rightarrow Set\) where

- \(vZ : Var\ (\Gamma , \sigma) (\sigma [ pop \sigma ]))\)
- \(vS : forall \tau \rightarrow Var \Gamma \sigma \rightarrow Var (\Gamma , \tau) (\sigma [ pop \tau ]))\)

Variables are defined as de Bruijn indices as they would be for simple types except their types must be weakened so that they are in the appropriate contexts. Their embedding operation is defined as follows:

- \(emb : Var \Gamma \sigma \rightarrow Tm \Gamma \sigma\)
- \(emb \ vZ = \text{top}\)
- \(emb \ (vS \tau \ x) = emb \ x \ [ \ pop \tau ]\)

Except for the more sophisticated treatment of types the only addition to the simply-typed definitions of values and neutral terms are the coercion constructors. As environments are just sequences of terms we can easily define coercion \(\text{coev}^v\) (mutually with coherence \(\text{coeh}^v\)) recursively and it does not play a role in the the actual definition of values so they can be defined separately.
Chapman

mutual
data Val : forall Γ -> Ty Γ -> Set where
  λv : Tm (Δ , σ) τ -> (vs : Env Δ) -> Val Γ (Π τ [ emb vs ])
  nev : NeV Γ σ -> Val Γ σ
  coev : Val Γ σ -> σ = σ' -> Val Γ' σ'

Values are either closures, neutral terms or coercions.

  emb : Val Γ σ -> Tm Γ σ
  emb (λv t vs) = λ t [ emb vs ]
  emb (nev n) = emb n
  emb (coev v p) = coe (emb v) p

Neutral terms are either variables, stuck applications or coercions.

data NeV : forall Γ -> Ty Γ -> Set where
  var : Var Γ s -> NeV Γ s
  app : {ts : Sub Γ Δ} -> NeV Γ (Π τ [ ts ]) ->
       (v : Val Γ (σ [ ts ])) -> NeV Γ (τ [ ts < emb v ])
  coen : NeV Γ σ -> σ = σ' -> NeV Γ' σ'

  emb : NeV Γ σ -> Tm Γ σ
  emb (var x) = emb x
  emb (app n v) = emb n $ emb v
  emb (coen n p) = coe (emb n) p

Environments are simple sequences of values.

data Env (Γ : Con) : Con -> Set where
  e = Env Γ e
  _<<_ : forall {Δ σ}(vs : Env Δ) -> Val Γ (σ [ emb vs ]) ->
         Env Γ (Δ , σ)

  emb : Env Γ Δ -> Sub Γ Δ
  emb (vs << v) = emb vs < emb v
  emb {Γ = e} e = id
  emb {Γ = Γ , σ} e = emb (emb {Γ}) • pop σ

Therefore identity environment must be defined recursively:

mutual
  vid : forall {Γ} -> Env Γ Γ
  vid {ε} = ...
  vid {Γ , σ} = ...

Due to the mutual definition of the environments and their embeddings each time
we define an operation that refers to environments (or values or neutral terms) we
must show that it interacts appropriately with embedding. This also means that
we are forced to completeness of the normaliser mutually with its definition. In the
case of the identity environment we require the following property which is proved
mutually with the definition of of vid:

8
The definitions that follow make heavy use of equational reasoning introduced by coercions. To make the definitions more readable most of the equality proof arguments to coercions have been replaced by 

Evaluation for terms $ev$, substitutions $ev^s$, and semantics application $vapp$ (of value functions to value arguments) are mutually defined. The use of syntactic types and explicit coercions forces us to define evaluation mutually with a coherence property: evaluating the term in an environment and then embedding it back into the syntax must give a term definitionally equal to the original term substituted by the environment embedded back into the syntax.

\[
\begin{align*}
\text{mutual} \\
\text{ev} &: \text{Tm} \Delta \sigma \rightarrow (vs : \text{Env} \Gamma \Delta) \rightarrow \text{Val} \Gamma (\sigma [\text{emb} \ vs]) \\
\text{ev} (\text{coe} \ t \ p) &: \text{vs} = \text{coev} (\text{ev} t (\text{coe}^s \ ts \ vs)) \ _ \\
\text{ev} (t [ ts ]) &: \text{vs} = \text{coev} (\text{ev} t (\text{ev}^s \ ts \ vs)) \ _ \\
\text{ev} \text{top} &: (vs << v) = \text{coev} v \ _ \\
\text{ev} (\lambda t) &: \text{vs} = \lambda v \ t \ vs \\
\text{ev} (\text{app} f) &: (vs << v) = \text{vapp} (\text{ev} f \ vs) \ \text{refl} \ v \\
\end{align*}
\]

The semantic application $vapp$ has a very liberal type which takes values whose types are equal to function types rather than actually are function types. This is necessary for the coercion case: The value $v$ in this case has an arbitrary type which is equal to a function type and it cannot be show at this stage that this must be a function type so instead we accumulate the coercions.

\[
\begin{align*}
\text{vapp} &: \{ts : \text{Sub} \ G \ D\} \rightarrow \text{Val} \ G' (r) \rightarrow r = (\Pi \sigma \tau [ts]) \rightarrow \\
&\hspace{1cm} (a : \text{Val} \ \Gamma (\sigma [ts])) \rightarrow \text{Val} \ \Gamma (\tau [ts < \text{emb} a]) \\
\text{vapp} (\lambda v \ t \ vs) &: p \ a = \text{coev} (\text{ev} t (vs << \text{coev} a \ _)) \ _ \\
\text{vapp} (\text{nev} n) &: p \ a = \text{nev} (\text{app} (\text{coen} n \ p) \ a) \\
\text{vapp} (\text{coe} v \ p) &: q \ a = \text{vapp} v (\text{trans} p \ q) \ a \\
\end{align*}
\]

The corresponding coherence properties are defined mutually:

\[
\begin{align*}
\text{cohev} &: (t : \text{Tm} \Delta \sigma)(vs : \text{Env} \Gamma \Delta) \rightarrow t [\text{emb} \ vs] = \text{emb} (\text{ev} t \ vs) \\
\text{cohev} = \ldots \\
\text{cohev}^s &: (ts : \text{Sub} \ G \ D)(vs : \text{Env} \Gamma \Delta) \rightarrow \\
&\hspace{1cm} ts \bullet \text{emb} vs = \text{emb} (\text{ev}^s \ ts \ vs) \\
\text{cohev}^s = \ldots \\
\text{cohvapp} &: \{ts : \text{Sub} \ G \ D\}(f : \text{Val} \ G' \rho)(p : \Pi \sigma \tau [ts]) \rightarrow \\
\end{align*}
\]
In the next section a type-directed quotation operation is required. For this reason
we need to define value types. Weak-head normal forms are exactly what is required
to perform type-directed operations as they tell you what the outer constructor is.
The definition of type values is quite simple. We have value versions of \( U \) and \( El \),
and \( \Pi \) is represented as a closure like \( \lambda \). We do not need a separete constructor for
value coersions as these can be pushed under \( VEl \) or into the environment.

\[
\text{data VTy : Con -> Set where}
\]
\[
\begin{align*}
\text{VU} &: \text{VTy} \\
\text{VEl} &: \text{Val} \text{ } U \to \text{VTy} \\
\text{yi} &: \forall \sigma \to \text{Ty} \text{ } (\Delta , \sigma) \to \text{Env} \text{ } \Delta \to \text{VTy} \\
\text{emb} &: \text{VTy} \to \text{Ty} \\
\text{emb} \text{ VU} &= U \\
\text{emb} \text{ (VEl } \sigma) &= \text{El} \text{ (emb } \sigma) \\
\text{emb} \text{ (yi } \sigma \tau \text{ vs)} &= \Pi \sigma \tau [ \text{emb vs} ]
\end{align*}
\]

Last is the definition of the evaluator for types and its coherence condition:

\[
\text{ev}^* : \text{Ty} \Delta \to \text{Env} \text{ } \Delta \to \text{VTy} \\
\text{ev}^* \text{ (coe } \sigma \text{ p)} \text{ vs} = \text{ev}^* \text{ } \sigma \text{ (coev}^* \text{ vs refl (sym p))} \\
\text{ev}^* \text{ (yi } \sigma \text{ [ ts ])} \text{ vs} = \text{ev}^* \text{ } \sigma \text{ (ev}^* \text{ ts vs)} \\
\text{ev}^* \text{ U} \text{ vs} = \text{VU} \\
\text{ev}^* \text{ (El } \sigma) \text{ vs} = \text{VEl} \text{ (coev } \text{ ev } \sigma \text{ vs) U[ ]} \\
\text{ev}^* \text{ (yi } \Pi \sigma \tau \text{ vs)} \text{ vs} = \text{yi } \sigma \tau \text{ vs}
\]

\[
\text{comev}^* : (\sigma : \text{Ty } \Delta)(\text{vs} : \text{Env } \Delta \to \sigma [ \text{emb vs } ] = \text{emb } (\text{ev}^* \sigma \text{ vs}) \\
\text{comev}^* = \ldots
\]

\section{Normal forms and quote}

The definition of \( \beta \eta \)-normal forms is defined mutually with neutral terms and again
with their corresponding embeddings back into the syntax. The types of the con-
structors \text{neu} and \text{neel} ensure that neutral terms are only appear at base type in
normal forms. We also require embeddings into values for the definition of quote but
these do not form part of the definition of normal forms so we omit them altogether.

\[
\text{mutual}
\]
\[
\text{data Nf : forall } \Gamma \to \text{Ty } \Gamma \to \text{Set where}
\]
\[
\begin{align*}
\text{ln} &: \text{Nf} (\Gamma , \sigma ) \tau \to \text{Nf} (\Pi \sigma \tau ) \\
\text{neu} &: \text{NeN } \Gamma U \to \text{Nf } \Gamma U \\
\text{neel} &: \text{NeN } \Gamma (\text{El } \sigma ) \to \text{Nf } \Gamma (\text{El } \sigma ) \\
\text{ncoe} &: \text{Nf } \Gamma \sigma \to \sigma = \sigma' \to \text{Nf } \Gamma' \sigma'
\end{align*}
\]

\[
\text{nemb} : \text{Nf } \Gamma \sigma \to \text{Tm } \Gamma \sigma
\]
data NeN : forall \Gamma \rightarrow Ty \Gamma \rightarrow Set where

  nvar : Var \Gamma \sigma \rightarrow NeN \Gamma \sigma
  napp : NeN \Gamma (\Pi \sigma \tau) \rightarrow (n : Nf \Gamma \sigma) \rightarrow NeN \Gamma (\text{sub} \tau (\text{nemb} n))
  ncoe : NeN \Gamma \sigma \rightarrow \sigma = \sigma' \rightarrow NeN \Gamma' \sigma'

nemb : NeN \Gamma \sigma \rightarrow \text{Tm} \Gamma \sigma
nemb = ...

As quotation is type directed and the values are indexed only by syntactic types an operation which replaces the type by (the result of embedding) its evaluated counterpart is required. As before the proof arguments to coercions are omitted and a coherence property is required.

mutual

  replace : Val \Gamma \sigma \rightarrow Val \Gamma (\text{emb} (\text{ev}^* \sigma \text{ vid}))
  replace (\lambda v t vs) = \lambda v t (\text{ev} (\text{emb} vs) \text{ vid})
  replace (nev n) = nev (nreplace n)
  replace (coev v p) = coev (replace v)

nreplace : NeV \Gamma \sigma \rightarrow NeV \Gamma (\text{emb} (\text{ev}^* \sigma \text{ vid}))

nreplace (var x) = coen (var x)

nreplace (app n v) =
  coen (app (nreplace n) (coev (replace v)))

nreplace (coen n p) = coen (nreplace n)

cohreplace : (v : Val \Gamma \sigma) \rightarrow \text{emb} (\text{replace} \ v) = \text{emb} \ v
cohreplace = ...

cohnreplace : (n : NeV \Gamma \sigma) \rightarrow \text{emb} (\text{nreplace} \ n) = \text{emb} \ n
cohnreplace = ...

The definition of quote takes a weak-head normal form and gives a \(\beta\eta\)-normal form. Quote for values is defined by (general) recursion on the type and is mutual with neutral quote which is defined by recursion on the structure of neutral terms. We require coherence properties for both:

mutual

  quote : (\sigma : VTy \Gamma) \rightarrow Val \Gamma (\text{emb} \ \sigma) \rightarrow Nf \Gamma (\text{emb} \ \sigma)
  quote (VI \sigma \tau vs) f =
    ncoe (\lambda n (quote
      (ev^* \tau (\text{ev}^* (\text{emb} vs) (\text{wk} \ [ \text{emb}^* vs ] ^*) \text{ vid})
      \langle < \text{coev} (nev (var \ vZ)) \rangle
      (\text{replace} (vapp (wk (\sigma [ \text{emb vs } ]) \text{ coev f })))
      \text{ refl}
      (nev (var vZ)))))

quote VU (nev n) = neu (quote n)
quote VU (vcoe \{ \sigma = \sigma \} v p) =
ncoe (quote (ev^+ σ vid) (replace v)) _
quote (VEl σ) (nev n) = neel (quote n)
quote (VEl σ) (vcoe {s = σ'} v p) =
  ncoe (quote (ev^+ σ' vid) (replace v)) _

cohquote : (σ : VTy Γ)(v : Val Γ (emb σ)) ->
  nemb (quote σ v) = emb v
cohquote σ v = ...

nquote : Ne Γ σ -> NeN Γ σ
nquote (var x) = nvar x
nquote (app {σ = σ}{ts = ts} n v) =
  ncoe (napp (ncoe (nquote n) P [[]])
  (ncoe (quote (ev^+ (σ [ ts ]) vid) (replace v)) _)) _
  nquote (coen n p) = ncoe (nquote n) p

cohnquote : (n : Ne Γ σ) -> nemb (nquote n) = emb n
cohnquote = ...

We can now define the normaliser and its coherence condition:

nf : Tm Γ σ -> Nf Γ (emb (ev^+ σ vid))
nf t = quote (replace (eval t vid))

cohnf : (t : Tm Γ σ) -> t = emb (nf t)
cohnf = ...

Notice that the coherence property for the normaliser is the usual completeness property for normalisation and follows from the coherence properties for eval, quote and replace.

5 Extension

We add codes for Π-types in the universe by adding a constructor to terms

Πu : forall (σ : Tm Γ U) -> Tm (Γ, El σ) U -> Tm Γ U

and the following rule to type equality:

ΠEl : El {Γ} (Πu' σ τ) = Π (El σ) (El τ)

The value type is also extended with a new constructor:

Πuv : forall (σ : Tm Δ U) -> Tm (Δ, El σ) U ->
  (vs : Env Γ Δ) -> Val Γ (U [ emb vs ])

This presents a new problem with the definition of semantic application vapp. It is defined by case on values of arbitrary type so there is a case vapp (Πuv σ τ vs) p a = ?. The equation p has the uninhabited type: U = Π σ' τ' but the type checker does not know that it is uninhabited. For this reason we must define an eliminator for this impossible equation in the syntax and carry it through to (neutral) values and (neutral) normal forms:
We must also define the following new equations:

\[
\begin{align*}
\text{botEl} & : (p : U \{\Gamma\} = P\{\Gamma\}) \Rightarrow (t : Tm \{\Gamma', \sigma\} \Rightarrow El\{\Gamma\} (\text{bot} p U)) = El (\text{coe} (\lambda t . \text{sym} p)) \\
\text{bot[]} & : \{ts : \text{Sub} \Gamma D\} \Rightarrow bot p \{ts\} = bot p (\rho \{ts\}) \\
\text{botapp} & : (p : U \{\Gamma\} = P\{\Gamma\}) \Rightarrow (a : Tm \{\Gamma', \sigma\}) \Rightarrow \text{coe} (\Pi \sigma \tau) p \ a = bot p (\Pi \sigma' \tau') \ a
\end{align*}
\]

To define quote we need a more sophisticated treatment of elements of the type \(\text{El} \sigma\):

We need a semantic decoder. We have to be more specific about neutral codes so we adapt the definition of value types to have a constructor for only neutral codes:

\[
\begin{align*}
\text{data VTy} & : \text{Con } \to \text{Set where} \\
\text{VU} & : \text{VTy} \Gamma \\
\text{NEL} & : \text{NeV} \Gamma U \Rightarrow \text{VTy} \Gamma \\
\text{V} & : \forall \sigma \Rightarrow \text{Ty} (\Delta, \sigma) \Rightarrow \text{Env} \Gamma \Delta \Rightarrow \text{VTy} \Gamma
\end{align*}
\]

The decoder turns coded \(\Pi\)-types into real ones and deals with neutral codes, coercions and the impossible case where the code is a \(\lambda\)-term.

\[
\begin{align*}
\text{decode} & : \forall \{ts : \text{Sub} \Gamma \Delta\} \Rightarrow \text{Val} \Gamma' \rho \Rightarrow \rho = U \{ts\} \Rightarrow \text{VTy} \Gamma \\
\text{decode} (\Pi \nu \sigma \tau \ vs) & = \Pi (\text{El} \sigma) (\text{El} \tau) (\text{coev}^\nu \ vs \ \_) \\
\text{decode} (\lambda v \ t \ vs) & = \text{NEL} (\text{botn} \ U) \\
\text{decode} (\text{nev} \ n) & = \text{NEL} (\text{coen} n \) \\
\text{decode} (\text{coev} v \ p) & = \text{decode} v (\text{trans} p \ q)
\end{align*}
\]

There is also a coherence condition which states that the decoded type is equal to the original. We must extend the evaluation, replacement and quote operations to deal with the new eliminator but these are trivial changes.

6 Conclusions

It is fair to say that this presentation is somewhat heavy. However we are able to present almost the complete code in this paper. We have only omitted things that can be easily recovered or could potentially be derived automatically.

Another (rather more philosophical) criticism of this work is: Why on earth would you write something that isn’t structurally recursive? It isn’t even a function! In the end we do intend to write a structurally recursive function. We will do this using the Bove-Capretta method[BC01] to define a new version of the normaliser which is structurally recursive on its own call graph. Deriving this new function from the original definition and defining the call graph is a tedious business and one that we might hope could be done automatically. Given that this is the case we emphasize the role of the original definition. Of course there is a step which we cannot do automatically and that is to prove termination which is what gives us an inhabitant of the call graph. Our intention here is to separate the different aspects of the constructive definition of a function. This gives a clear advantage: We can
present just one aspect in its entirety. Our development to this point is about 1000
lines and we have presented all but the most laborious aspects. Danielsson’s NBE
normaliser is an order of magnitude larger and presented in a similar programming
language. Our completed development is likely to be of similar size but we are able
to isolate an informative fragment and present it in a self contained way.

The fragment we isolate is the runtime or computational behaviour. It is interesting
that if one takes a function defined using the Bove-Capretta technique and
compiles it using Brady’s techniques [Bra05] (erasing runtime-irrelevant information)
one ends up with exactly the original function.

Another way of looking at our definition is to say that it is not the definition
of a normaliser but instead it is the specification of one. We have not defined an
evaluator but instead given some laws which an evaluator should obey. In [ACD08]
an algebraic description of values is given. In their case values are a syntactic
applicative structure with an evaluation operation. The axioms that must hold for
evaluation look suspiciously like our definitions of evaluators. As their notion of
values is untyped their definition bears a closer resemblance to our simply-typed
case than the one presented here.

It can be said of type theory and constructive mathematics in general that they
take computation as the most fundamental concept. More fundamental than either
writing a proof or a program is the concept that underlies both of them: that of an
algorithm. Normalisation by evaluation provides the semantics for type theory most
easily reconcilable with the constructive notion of computation. It explains type
theory using its own notion of computation that of function definition by structural
recursion. More traditional normalisation proofs based on small-step reduction
cannot avoid extolling the view that type theory’s notion of computation is in some
way inadequate and that reduction is more fundamental. Big-step normalisation
could be seen to represent a similar position. In our case it is the idea of computation
by an environment machine which we take to be more fundamental. Having said
this our method (by not stopping at just giving a big-step semantics and proving
that it terminates) reconciles this notion of computation with the more constructive
one in the end by adding termination information and it is our hope that by doing
this we do not have to reject definition by structural recursion as the notion of
computation which we hold most dear.

6.1 Future and ongoing work

Extending this presentation to include natural numbers and formalizing termination
and soundness of this program and its extensions are further work. I have been
rather timid about induction-recursion. I have used it only for embeddings and
these are defined by recursion on the mutually defined inductive types. In fact even
the mutual inductive definitions presented here (e.g. the presentation of the syntax)
fall outside Dybjer’s schemas for inductive families [Dyb91].

6.2 Acknowledgments

I would like to thank Nils Anders Danielsson, Thorsten Altenkirch, Wouter Swiestra,
Nicolas Oury and the anonymous referees for their helpful comments.
Chapman

References


