Available online at www.sciencedirect.com
ScienceDirect

Electronic Notes in Theoretical Computer Science 228 (2009) 21–36

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Type Theory Should Eat Itself

James Chapman[1]

School of Computer Science
University of Nottingham
Jubilee Campus
Nottingham, NG8 1BB, England

Abstract

In this paper I present a partial formalisation of a normaliser for type theory in Agda[22]; extending previous work on big-step normalisation[2,1]. The normaliser in 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[23] and semantics[15,5] 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[8]. 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"[21]. 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[2,1] and extend it to dependent types. The approach is as follows: 1. Define the well-typed terms of the

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

doi:10.1016/j.entcs.2008.12.114

language; 2. Give a simple (but not obviously terminating) normaliser as an environment machine; 3. Encode the normaliser as a big-step semantics and 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[15,5] has concerned internalising intuitionistic model theory and proofs of normalisation by evaluation (NBE)[7]. Recently Danielsson[11] 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 developement 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[13] 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[15,17] and Martin-Löf's substitution calculus[20]. We diverge from the usual presentation of internal categories with families[15] 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[23] and proved a number of properties but not normalisation. Barras and Werner have formalised the Calculus of Constructions in Coq[8] 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[16].

1.2 Agda and notation

Agda is a dependently typed programming language. It supports inductive families[14] and dependent pattern matching[10]. 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 `_+_` and when used must be surrounded by space `m + n`. Implicit arguments (arguments which can be inferred from use) are given in braces `{a : A}`. When they cannot be inferred we supply them in braces `f {B} a` in the appropriate place or by name `f {B = B} a`. The keyword `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[9]. We use implicit quantification to omit implicit arguments in definitions of constructors. An example of this is the identity substitution which we write as `id : Sub Γ Γ` when in current syntax we would have to write `id : forall {Γ} -> Sub Γ Γ`. 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[17] 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 Π-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:

```
Con : Set                           -- Context
Ty  : Con -> Set                    -- Type
Tm  : forall Γ -> Ty Γ -> Set       -- Term
Sub : Con -> Con -> Set             -- Substitution
```

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 `coe`, we cannot separate the equation syntax from the syntax of well-typed terms. Hence the corresponding equality judgments must also be mutually defined:

```
_=_ : Con -> Con -> Set             -- Equality of contexts
_=_ : Ty Γ -> Ty Γ' -> Set          -- Equality of types
_=_ : Tm Γ σ -> Tm Γ' σ' -> Set     -- Equality of terms
_=_ : Sub Γ Δ -> Sub Γ' Δ' -> Set   -- Equality of substitutions
```

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[12] 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
  data Con : Set where
    ε   : Con
    _,_ : forall Γ -> Ty Γ -> Con
```

Types are indexed by contexts:

```
  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 Γ and Γ and Δ are equal contexts then we have a type in Δ.

```
    coe   : Ty Γ -> Γ = Δ -> Ty Δ
```

There is a constructor for explicit substitutions.

```
    _[_]  : Ty Δ -> Sub Γ Δ -> Ty Γ
```

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 Π-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 (Π-types). Notice that the second argument (the range) to Π contains has a extra variable in its context (the domain).

```
    U     : Ty Γ
    El    : Tm Γ U -> Ty Γ
    Π     : (σ : Ty Γ) -> Ty (Γ , σ) -> Ty Γ
```

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

```
  data Tm : forall Γ -> Ty Γ -> Set where
    coe  : Tm Γ σ -> σ = σ' -> Tm Γ' σ'
    _[_] : Tm Δ σ -> (ts : Sub Γ Δ) -> Tm Γ (σ [ ts ])
```

Variables (de Bruijn indices[12]) are not singled out as a separate syntactic class in the syntax. Instead we have `top` which is the first bound variable and then other variables are obtained by applications of the weakening substitution (which is called `pop` and is introduced ed below) (`top [ pop σ ]`,`top [ pop σ • pop τ ]`, etc.) to `top`. This treatment of variables is present in categorical treatments of type theory such as categories with families[17,14]

```
  top  : Tm (Γ , σ) (σ [ pop σ ])
```

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.

```
  λ    : Tm (Γ , σ) τ -> Tm Γ (Π σ τ)
  ap   : Tm Γ (Π σ τ) -> Tm (Γ , σ) τ
```

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 Sub : Con -> Con -> Set where
  coe : Sub Γ Δ -> Γ = Γ' -> Δ = Δ' -> Sub Γ' Δ'
  id  : Sub Γ Γ
  pop : forall σ -> Sub (Γ , σ) Γ
  _<_ : (ts : Sub Γ Δ) -> Tm Γ (σ [ ts ]) -> Sub Γ (Δ , σ)
  _•_ : Sub Γ Δ -> Sub B Γ -> Sub B Δ
```

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 `El` to a term whose type is the constant `U` applied to a substitution without using a coercion directly. Whilst this type is equal to `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 `pop` and `top` directly.

```
sub⁺ : Ty (Γ , σ) -> Tm Γ σ -> Ty Γ
sub  : Tm (Γ , σ) τ -> (a : Tm Γ σ) -> Tm Γ (sub⁺ τ a)
Elˢ  : {ts : Sub Γ Δ} -> Tm Γ (U [ ts ]) -> Ty Γ
_$_  : Tm Γ (Π σ τ) -> (a : Tm Γ σ) -> Tm Γ (sub⁺ τ a)
_$ˢ_ : {ts : Sub Γ Δ} -> Tm Γ (Π σ τ [ ts ]) ->
        (a : Tm Γ (σ [ ts ])) -> Tm Γ (τ [ ts < a ])
_↗_  : (ts : Sub Γ Δ)(σ : Ty Δ) -> Sub (Γ , σ [ ts ]) (Δ , σ)
```

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 Con.

```
data  _=_ : Con -> Con -> Set where
```

Context equality is not always included in presentations of type theory and of its models: Hofmann[17] and Streicher[24] include it; Dybjer does not[15]; and Martin-Löf omits it from his presentations of type theory[19,21] but includes it in the substitution calculus[20].

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[18].

```
data  _=_ : Ty Γ -> Ty Γ’ -> Set where
```

The type level coercion induces a coherence condition:

```
  coh : (σ : Ty Γ)(p : Γ = Δ) -> coe σ p = σ
```

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

```
  rid   : {σ : Ty Γ} -> σ [ id ] = σ
  assoc : {ts : Sub Γ Δ}{us : Sub B Γ} ->
          σ [ ts ] [ us ]  = σ [ ts • us ]
  U[]   : {ts : Sub Γ Δ} -> U [ ts ] = U {Γ}
  El[]  : {t : Tm Δ U}{ts : Sub Γ Δ} ->
          El t [ ts ] = El^s (t [ ts ])
  Π[]   : {ts : Sub Γ Δ} ->
          Π σ τ [ ts ] = Π (σ [ ts ]) (τ [ ts ↗ σ ])
```

Semantic application requires projection from equations between Π-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.

```
  dom : {τ : Ty (Γ , σ)}{τ’ : Ty (Γ’ , σ’)} ->
        Π σ τ = Π σ’ τ’ -> σ = σ’
  cod : {τ : Ty (Γ , σ)}{τ’ : Ty (Γ’ , σ’)} ->
        Π σ τ = Π σ’ τ’ -> τ = τ’
```

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: β,η and projection from a substitution:

```
 data _=_ : Tm Γ σ -> Tm Γ' σ' -> Set where
    β    : {t : Tm (Γ , σ) τ} -> ap (λ t) = t
    η    : {f : Tm Γ (Π σ τ)} -> λ (ap f) = f
    top< : {ts : Sub Γ Δ}{t : Tm Γ (σ [ ts ])} -> top [ ts < t ] = t
```

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

```
 data _=_ : Sub Γ Δ -> Sub Γ' Δ' -> Set where
    lid    : {ts : Sub Γ Δ} -> id • ts = ts
    pop<   : {ts : Sub Γ Δ}{t : Tm Γ (σ [ ts ])} ->
             pop σ • (ts < t) = ts
    •<     : {ts : Sub Γ Δ}{t : Tm Γ (σ [ ts ])}{us : Sub B Γ} ->
             (ts < t) • us = (ts • us < coe (t [ us ]) assoc)
    poptop : (pop σ < top) = id {Γ , σ}
```

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
  fog : {σ : Ty Γ}{σ' : Ty Γ'} -> σ = σ' -> Γ = Γ'
  fog : {t : Tm Γ σ}{t' : Tm Γ' σ'} -> t = t' -> σ = σ'
  fog : {ts : Sub Γ Δ}{ts' : Sub Γ' Δ'} -> ts = ts' -> Γ = Γ'
```

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[2]. 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 Γ -> Ty Γ -> Set where
  vZ : Var (Γ , σ) (σ [ pop σ ])
  vS : forall τ -> Var Γ σ -> Var (Γ , τ) (σ [ pop τ ])
```

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 Γ σ -> Tm Γ σ
emb vZ       = top
emb (vS τ x) = emb x [ pop τ ]
```

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 `coev`^{s} (mutually with coherence `coeh`^{s}) recursively and it does not play a role in the the actual definition of values so they can be defined separately.

```
mutual
  data Val : forall Γ -> Ty Γ -> Set where
    λv    : Tm (Δ , σ) τ -> (vs : Env Γ Δ) ->
            Val Γ (Π σ 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 σ -> NeV Γ σ
    app  : {ts : Sub Γ Δ} -> NeV Γ (Π σ 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 $^s 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 ε
    _<<_ : forall {Δ σ}(vs : Env Γ 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 = ε}     e         = id
  emb {Γ = Γ , σ} e         = emb e {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:

```
  cohvid : forall {Γ} -> id {Γ} = emb (vid {Γ})
  cohvid = ...
```

The definitions that follow makes 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.

```
mutual
  ev : Tm Δ σ -> (vs : Env Γ Δ) -> Val Γ (σ [ emb vs ])
  ev (coe t p)     vs        = coev (ev t (coev^s vs _ _)) _
  ev (t [ ts ])    vs        = coev (ev t (ev^s ts vs)) _
  ev top           (vs << v) = coev v _
  ev (λ t)         vs        = λv t vs
  ev (app f)       (vs << v) = vapp (ev f vs) refl v

  ev^s : Sub Δ Σ -> Env Γ Δ -> Env Γ Σ
  ev^s (coe^s ts p q) vs        = coev^s (ev^s ts (coev^s vs _ _)) _ _
  ev^s (ts • us)      vs        = ev^s ts (ev^s us vs)
  ev^s id             vs        = vs
  ev^s (pop σ)        (vs << v) = vs
  ev^s (ts < t)       vs        = ev^s ts vs << coev (ev t vs) _
```

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.

```
  vapp : {ts : Sub Γ Δ} -> Val Γ' ρ -> ρ = (Π σ τ [ ts ]) ->
         (a : Val Γ (σ [ ts ])) -> Val Γ (τ [ ts < emb a ])
  vapp (λv t vs)  p a = coev (ev t (vs << coev a _)) _
  vapp (nev n)    p a = nev (app (coen n p) a)
  vapp (coev v p) q a = vapp v (trans p q) a
```

The corresponding coherence properties are defined mutually:

```
  cohev : (t : Tm Δ σ)(vs : Env Γ Δ) -> t [ emb vs ] = emb (ev t vs)
  cohev = ...

  cohev^s : (ts : Sub Δ Σ)(vs : Env Γ Δ) ->
            ts • emb vs = emb (ev^s ts vs)
  cohev^s = ...

  cohvapp : {ts : Sub Γ Δ}(f : Val Γ' ρ)(p : ρ = Π σ τ [ ts ]) ->
            (v : Val Γ (σ [ ts ])) ->
            coe (emb f) p $^s emb v = emb (vapp f p v)
  cohvapp = ...
```

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 Π is represented as a closure like λ. We do not need a seperate constructor for value coersions as these can be pushed under `VEl` or into the environment.

```
data VTy : Con -> Set where
  VU  : VTy Γ
  VEl : Val Γ U -> VTy Γ
  VΠ  : forall σ -> Ty (Δ , σ) -> Env Γ Δ -> VTy Γ

emb : VTy Γ -> Ty Γ
emb VU           = U
emb (VEl σ)      = El (emb σ)
emb (VΠ σ τ vs) = Π σ τ [ emb vs ]
```

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

```
ev^+ : Ty Δ -> Env Γ Δ -> VTy Γ
ev^+ (coe σ p)  vs = ev^+ σ (coev^s vs refl (sym p))
ev^+ (σ [ ts ]) vs = ev^+ σ (ev^s ts vs)
ev^+ U          vs = VU
ev^+ (El σ)     vs = VEl (coev (ev σ vs) U[])
ev^+ (Π σ τ)    vs = VΠ σ τ vs
```

```
comev⁺ : (σ : Ty Δ)(vs : Env Γ Δ) -> σ [ emb vs ] = emb (ev⁺ σ vs)
comev⁺ = ...
```

4 Normal forms and quote

The definition of βη-normal forms is defined mutually with neutral terms and again with their corresponding embeddings back into the syntax. The types of the constructors `neu` and `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.

```
mutual
  data Nf : forall Γ -> Ty Γ -> Set where
    λn   : Nf (Γ , σ) τ -> Nf Γ (Π σ τ)
    neu  : NeN Γ U -> Nf Γ U
    neel : NeN Γ (El σ) -> Nf Γ (El σ)
    ncoe : Nf Γ σ -> σ = σ' -> Nf Γ' σ'

  nemb : Nf Γ σ -> Tm Γ σ
  nemb = ...

  data NeN : forall Γ -> Ty Γ -> Set where
    nvar : Var Γ σ -> NeN Γ σ
    napp : NeN Γ (Π σ τ) -> (n : Nf Γ σ) -> NeN Γ (sub τ (nemb n))
    ncoe : NeN Γ σ -> σ = σ' -> NeN Γ' σ'

  nemb : NeN Γ σ -> Tm Γ σ
  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 Γ σ -> Val Γ (emb (ev⁺ σ vid))
  replace (λv t vs)  = λv t (ev (emb vs) vid)
  replace (nev n)    = nev (nreplace n)
  replace (coev v p) = coev (replace v) _

  nreplace : NeV Γ σ -> NeV Γ (emb (ev⁺ σ 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 Γ σ) -> emb (replace v) = emb v
  cohreplace = ...
  cohnreplace : (n : NeV Γ σ) -> emb (nreplace n) = emb n
  cohnreplace = ...
```

The definition of quote takes a weak-head normal form and gives a βη-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 : (σ : VTy Γ) -> Val Γ (emb σ) -> Nf Γ (emb σ)
  quote (VΠ σ τ vs) f =
    ncoe (λn (quote
                 (ev^+ τ (ev^s (emb vs) (wk^s [ emb^s vs ]^+) vid)
                        << coev (nev (var vZ) _))
                 (replace (vapp (wk (σ [ emb vs ]) (coev f _))
                               refl
                               (nev (var vZ))))))

         -
  quote VU          (nev n)             = neu (quote n)
  quote VU          (vcoe {σ = σ} v p)  =
    ncoe (quote (ev^+ σ vid) (replace v)) _
  quote (VEl σ)     (nev n)             = neel (quote n)
  quote (VEl σ)     (vcoe {σ = 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) Π[])
               (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:

```
  bot    : U {Γ} = Π {Γ'} σ τ (ρ : Ty Γ'') -> Tm Γ'' ρ
  botn   : U {Γ} = Π {Γ'} σ τ (ρ : Ty Γ'') -> NeV Γ'' ρ
  nbot   : U {Γ} = Π {Γ'} σ τ (ρ : Ty Γ'') -> NeN Γ'' ρ
```

We must also define the following new equations:

```
  botEl  : (p : U {Γ} = Π {Γ'} σ τ){t : Tm (Γ' , σ) τ} ->
            El {Γ} (bot p U) = El (coe (λ t) (sym p))
  bot[]  : {ts : Sub Γ Δ}{p : U {Γ'} = Π {Γ''} σ τ} ->
            bot p ρ [ ts ] = bot p (ρ [ ts ])
  botapp : (p : U {Γ} = Π {Γ'} σ' τ'){a : Tm Γ' σ'} ->
           coe (Πu σ τ) p $ a = bot p (Π σ' τ') $ a
```

To define quote we need a more sophisticated treatment of elements of the type El σ: 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:

```
data VTy : Con -> Set where
  VU  : VTy Γ
  NEl : NeV Γ U -> VTy Γ
  VΠ  : forall σ -> Ty (Δ , σ) -> Env Γ Δ -> VTy Γ
```

The decoder turns coded Π-types into real ones and deals with neutral codes, coercions and the impossible case where the code is a λ-term.

```
decode : forall {ts : Sub Γ Δ} -> Val Γ' ρ -> ρ = U [ ts ] -> VTy Γ
decode (Πuv σ τ vs) p = VΠ (El σ) (El τ) (coev^s vs _ _)
decode (λv t vs)    p = NEl (botn _ U)
decode (nev n)      p = NEl (coen n _)
decode (coev v p)   q = decode v (trans p q)
```

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[4] 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[6] (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 [3] 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[14].

Acknowledgement

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

References

[1] Thorsten Altenkirch and James Chapman. Tait in one big step. In *Workshop on Mathematically Structured Functional Programming, MSFP 2006, Kuressaare, Estonia, July 2, 2006*, electronic Workshop in Computing (eWiC), Kuressaare, Estonia, 2006. The British Computer Society (BCS).

[2] Thorsten Altenkirch and James Chapman. Big-Step Normalisation. *Journal of Functional Programming*, 2008. Special Issue on Mathematically Structured Functional Programming. To appear.

[3] Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory. In *Mathematics of Program Construction, MPC'08*, 2008. To appear.

[4] Ana Bove and Venanzio Capretta. Nested General Recursion and Partiality in Type Theory. In Richard Boulton and Paul Jackson, editor, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2001.

[5] Alexandre Buisse and Peter Dybjer. Towards formalizing categorical models of type theory in type theory. In *Proc. Int. Workshop on Logical Frameworks and Meta-Languages (LFMTP'07)*, volume 196 of *Electronic Notes in Computer Science*, pages 137–151, 2008.

[6] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language.* PhD thesis, University of Durham, 2005.

[7] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, 1991.

[8] Bruno Barras and Benjamin Werner. Coq in Coq. unpublished.

[9] James Chapman. Formalisation of Big-Step Normalisation in Agda, 2007. Available from http://www.cs.nott.ac.uk/~jmc/BSN.html.

[10] Thierry Coquand. Pattern Matching with Dependent Types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Electronic Proceedings of the Third Annual BRA Workshop on Logical Frameworks (Båstad, Sweden)*, 1992.

[11] Nils Anders Danielsson. A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, Nottingham, England, April 2006. Springer-Verlag LNCS 4502.

[12] Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.

[13] Peter Dybjer and Anton Setzer. Indexed induction-recursion. In Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2001.

[14] Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.

[15] Peter Dybjer. Internal Type Theory. In *Types for Proofs and Programs, '95*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[16] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994.

[17] Martin Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, volume 14, pages 79–130. Cambridge University Press, Cambridge, 1997.

[18] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1999.

[19] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

[20] Per Martin-Löf. Substitution calculus, 1992. Notes from a lecture given in Göteborg.

[21] Per Martin-Löf. An intuitionistic theory of types. In G. Sambin and J. Smith, editors, *Twenty-five years of constructive type theory*, volume 36 of *Oxford Logic Guides*. Oxford University Press, 1998.

[22] Ulf Norell. Agda 2, 2007. http://www.cs.chalmers.se/~ulfn/.

[23] Robert Pollack. *The Theory of LEGO*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1995.

[24] Thomas Streicher. *Semantics of type theory: correctness, completeness, and independence results*. Birkhäuser Boston Inc., Cambridge, MA, USA, 1991.