Hereditary Substitutions for Simple Types, Formalized

Chantal Keller *
INRIA Saclay–Île-de-France
France
keller@lix.polytechnique.fr

Thorsten Altenkirch
The University of Nottingham
United Kingdom
txa@cs.nott.ac.uk

Abstract

We analyze a normalization function for the simply typed λ-calculus based on hereditary substitutions, a technique developed by Pfenning et al. The normalizer is implemented in Agda, a total language where all programs terminate. It requires no termination proof since it is structurally recursive which is recognized by Agda's termination checker. Using Agda as an interactive theorem prover we establish that our normalization function precisely identifies $\beta\eta$-equivalent terms and hence can be used to decide $\beta\eta$-equality. An interesting feature of this approach is that it is clear from the construction that $\beta\eta$-equality is primitive recursive.

Keywords Hereditary substitutions, Type Theory, normalizer, decidability of $\beta\eta$-equality

1. Introduction

Among the different ways to establish the decidability of some equality for λ-calculi (e.g. see [6, 16]), it seems natural in a programming point of view to give a normalization function, and to prove that it computes a canonical form for any term. This approach is well-suited for an implementation, and it thus becomes important to be able to analyze it in order to prove, for instance, termination of the process.

In this paper, we define a normalizer that β-reduces and η-expands simply typed λ-terms. We use it to establish a formal verification of the decidability of the $\beta\eta$-equality for this calculus.

The normalizer and the proofs are implemented in Agda [1, 13]. Agda is a programming language with dependent types. We exploit this property in two ways. First, the typing system is powerful enough to ensure some non trivial properties of the functions defined in this language. Secondly, we can use Agda as an interactive theorem prover.

Thus, Agda is our metalanguage all along the paper. The source code can be found online [2].

This paper explains our implementation choices and gives the main ideas of the proof of decidability of the $\beta\eta$-equality.

We consider a typed syntax for λ-calculus, that is to say we consider only well-typed terms (Section 2). It uses De Bruijn indices, which are adapted to a formal development since closed terms have a unique form (Section 2.2).

The normalizer implements hereditary substitutions (Section 3) [17], which preserve **canonical forms** by substituting and reducing the redices that can appear from substitution at the same time. An important aspect of hereditary substitution is that it can be defined using structural induction; hence, it can be implemented in total Type Theory without any termination proof. In our development, Agda's termination checker is able to check that the normalizer does terminate (Section 3.4).

We prove the completeness (Section 4) and the soundness (Section 5) of our normalizer, in order to conclude that it decides the $\beta\eta$-equivalence of two terms.

We finally discuss related work (Section 6) and conclude (Section 7).

2. The simply typed λ-calculus

We start by introducing our calculus. Its particularity is that we only define well typed terms: terms are objects of an inductive family parameterized by types. This presentation of the simply typed λ-calculus is formally described in [7], for instance. It is very convenient to use in Agda, which supports dependent type programming and inductive declarations.

2.1 The calculus

The set of types (Ty) is defined by a simple inductive definition:

```
data Ty : Set where
  ∘ : Ty
  _⇒_ : Ty → Ty → Ty
```

Inductive definitions are introduced in Agda with the keywords **data** and **where**. Each following line defines a new constructor of this inductive type, giving its type. Set is Agda's type of types, and the _•_ notation is used to define mixfix operators.

For terms, we use (typed) De Bruijn indices to represent variables [11]. We thus need a context that maps each free variable to a type. Since we do not have to store names, a context (Con) can be represented as a list of types:

```
data Con : Set where
  ε : Con
  _,_ : Con → Ty → Con
```

As explained above, the sets of variables (Var) and terms (Tm) are objects of an inductive family indexed by types and contexts:

```
data Var : Con → Ty → Set where
  vz : forall {Γ σ} → Var (Γ, σ) σ
  vs : forall {τ Γ σ} → Var Γ σ → Var (Γ, τ) σ
data Tm : Con → Ty → Set where
  var : forall {Γ σ} → Var Γ σ → Tm Γ σ
```

* Secondary affiliation: École Normale Supérieure de Lyon, France

```
Λ : forall {Γ σ τ} → Tm (Γ,σ) τ → Tm Γ (σ ⇒ τ)
app : forall {Γ σ τ} →
  Tm Γ (σ ⇒ τ) → Tm Γ σ → Tm Γ τ
```

In Agda, curly brackets surround the *implicit arguments* of a constructor or a function. Usually, these arguments can be automatically infered by Agda, so they do not have to be given when the constructor or the function is applied. For a matter of space, in the paper, we skip them also in type declarations when it does not lead to any ambiguity: we consider that any symbol appearing in the type of a constructor or a function that is not previously defined is an implicit argument. For instance, the type of app would now be written:

```
app : Tm Γ (σ ⇒ τ) → Tm Γ σ → Tm Γ τ
```

2.2 Working with De Bruijn indices

De Bruijn indices have the nice property to give a unique representation to any closed term, which makes them easy to implement. They are not natural to use for a human being, though. For clarity, this section presents some tools to work with De Bruijn indices in our calculus.

2.2.1 Vocabulary

First, we use a rigid vocabulary all along the paper:

- If x: Var Γ σ, we call x an *index*.
- If x: Var Γ σ, we say that x is *parameterized* by Γ.
- We say that x and y *represent* the same variable if they would have the same name in a named representation, but are not parameterized by the same context.

For instance, the λ-term:

$$\lambda f.\lambda z.f\ ((\lambda x.f\ x)\ z)$$

is encoded in Agda by

```
Λ (Λ (app (var (vs vz))
  (app (Λ (app (var (vs (vs vz))) (var vz))) (var vz))))
```

In this term, the two occurrences of var vz **do not** represent the same variable (it can be x or z), whereas vs vz and vs (vs vz) **do** represent the same variable (f).

2.2.2 Removal from a context

In our calculus, many constructions rely on removing an index parameterized by a context Γ from Γ. This is made possible by the following function:

```
_-_ : {σ : Ty} → (Γ : Con) → Var Γ σ → Con
ε - ()
(Γ,σ) - vz = Γ
(Γ,τ) - (vs x) = (Γ - x),τ
```

In Agda, functions are defined in a Haskell-like syntax: the first line gives the type of the function, and the other lines give its definition with (possibly) a case analysis. The () notation is useful to discriminate absurd cases: here, it is impossible for an index to be parameterized by the empty context ε.

2.2.3 Weakening

In a calculus with contexts and De Bruijn indices, weakening is also a very standard construction: it is often used to avoid capture. It means adding extra-information in a context parameterizing an index:

```
wkv : (x : Var Γ σ) → Var (Γ - x) τ → Var Γ τ
wkv vz y = vs y
wkv (vs x) vz = vz
wkv (vs x) (vs y) = vs (wkv x y)
```

2.2.4 Treatment of variable equality

We established in section 2.2.1 that the same index can represent different variables and, conversely, the same variable can be represented by different indices. Comparing variables is thus non trivial.

We introduce a predicate EqV specifying if two indices parameterized by the same context represent the same variable or not. Its return value is more precise than a boolean: it does not only answer "yes" or "no", but also gives additional information to justify it.

Its definition is the following one:

```
data EqV : Var Γ σ → Var Γ τ → Set where
  same : {x : Var Γ σ} → EqV x x
  diff : (x : Var Γ σ) → (y : Var (Γ - x) τ) →
    EqV x (wkv x y)
```

It relies on two properties:

1. The only way for x and y to represent the same variable is to be equal.
2. If x and y do not represent the same variable, then there exists an index z such that x ≡ wkv y z (≡ stands for Leibniz equality, with the three usual functions refl, sym and trans). The intuition behind this property is that if x and y do not represent the same variable, then x "exists" in Γ - y: this is the index z. x is thus the weakening of z when y is added to the context.

To summarize, for any variables x and y, EqV x y is inhabited by:

- same if x ≡ y;
- diff x z if y ≡ wkv x z.

The function eq decides EqV:

```
eq : (x : Var Γ σ) → (y : Var Γ τ) → EqV x y
eq vz vz = same
eq vz (vs x) = diff vz x
eq (vs x) vz = diff (vs x) vz
eq (vs x) (vs y) with eq x y
eq (vs x) (vs .x) | same = same
eq (vs .x) (vs .(wkv x y)) | (diff x y) = diff (vs x) (vs y)
```

The **with** construction allows pattern matching on a recursively computed result. Dot patterns (.x for instance) tag Agda terms whose construction is constrained by the value of the predicate.

2.3 Term weakening

Weakening is also defined for terms, if we add an index into the De Bruijn context parameterizing them:

```
wkTm : (x : Var Γ σ) → Tm (Γ - x) τ → Tm Γ τ
wkTm x (var v) = var (wkv x v)
wkTm x (Λ t) = Λ (wkTm (vs x) t)
wkTm x (app t₁ t₂) = app (wkTm x t₁) (wkTm x t₂)
```

2.4 The substitution function

The substitution function substitutes all occurrences of a free variable x in some term t by another term u. The type we give to the substitution function is: (t : Tm Γ τ) → (x : Var Γ σ) → (u : Tm (Γ - x) σ) → Tm (Γ - x) τ. Agda's typing ensures two fundamental properties of such a substitution:

1. Substitution is *type preserving*: since x has the same type (σ) as u, the type of t (τ) is preserved.

2. Since the result is parameterized by Γ - x, we know that x *does not appear free* in it.

Variable equality defined in section 2.2.4 plays a main role in substitution. Indeed, in the case where t is an index y, we need to know whether x and y represent the same variable or not.

We define the substitution for variables:

```
substVar : Var Γ τ → (x : Var Γ σ) →
  Tm (Γ - x) σ → Tm (Γ - x) τ
substVar v x u with eq x v
substVar v .v u | same = u
substVar . (wkv v x) .v u | diff v x = var x
```

and for terms:

```
subst : Tm Γ τ → (x : Var Γ σ) →
  Tm (Γ - x) σ → Tm (Γ - x) τ
subst (var v) x u = substVar v x u
subst (Λ t) x u = Λ (subst t (vs x) (wkTm vz u))
subst (app t₁ t₂) x u = app (subst t₁ x u) (subst t₂ x u)
```

2.5 Convertibility

The conversion relation we consider in this paper is the $\beta\eta$-equivalence ($\beta\eta$-$\equiv$), defined as an inductive predicate:

```
data _βη-≡_ : Tm Γ σ → Tm Γ σ → Set where
  brefl : {t : Tm Γ σ} → t βη-≡ t
  bsym : t₁ βη-≡ t₂ → t₂ βη-≡ t₁
  btrans : t₁ βη-≡ t₂ → t₂ βη-≡ t₃ → t₁ βη-≡ t₃
  congΛ : t₁ βη-≡ t₂ → Λ t₁ βη-≡ Λ t₂
  congApp : t₁ βη-≡ t₂ → u₁ βη-≡ u₂ →
    app t₁ u₁ βη-≡ app t₂ u₂
  beta : app (Λ t) u βη-≡ subst t vz u
  eta : Λ (app (wkTm vz t) (var vz)) βη-≡ t
```

It is an equivalence (brefl, bsym and btrans) that is congruent with the constructors of λ-terms (congΛ and congApp), and that identifies terms differing from one step of β-reduction (beta) or one step of η-expansion (eta).

3. Normalization and hereditary substitutions

We recall our goal is to show that the conversion relation we have just defined is decidable. We are going to establish this by normalization.

3.1 Normal forms

We first define the set of normal forms (Nf). In our context, normal forms are:

- neutral terms (Ne, ne): variables applied to as many normal arguments as their "arity". Lists of such arguments are called spines Sp and are parameterized by two types:
  1. The first type refers to the type of the variable.
  2. The second type refers to the resulting type of the application.

  Since we define long $\beta\eta$-normal forms, a neutral term is normal only if its type is $\circ$.
- λ-abstractions (λn).

```
mutual
  data Nf : Con → Ty → Set where
    λn : Nf (Γ, σ) τ → Nf Γ (σ ⇒ τ)
    ne : Ne Γ ∘ → Nf Γ ∘
  data Ne : Con → Ty → Set where
    _,_ : Var Γ σ → Sp Γ σ τ → Ne Γ τ
  data Sp : Con → Ty → Ty → Set where
    ε : Sp Γ σ σ
    _,_ : Nf Γ τ → Sp Γ σ ρ → Sp Γ (τ ⇒ σ) ρ
```

Here, the set of normal forms is not defined as a subset of the set of terms. However, it can be easily seen as such through the canonical injection ($\lceil \_ \rceil$):

```
mutual
  ⌈_⌉ : Nf Γ σ → Tm Γ σ
  ⌈ λn n ⌉ = Λ ⌈ n ⌉
  ⌈ ne n ⌉ = embNe n
  embNe : Ne Γ σ → Tm Γ σ
  embNe (v, s) = embSp s (var v)
  embSp : Sp Γ σ τ → Tm Γ σ → Tm Γ τ
  embSp ε acc = acc
  embSp (n, s) acc = embSp s (app acc ⌈ n ⌉)
```

Note that the function embSp, that maps spines into terms, is defined using an accumulator.

We are now going to define a normalization function nf : Tm Γ σ → Nf Γ σ, that maps each term to a normal representative. This representative corresponds to the *semantics* given to the initial term. In order to establish the decidability of the $\beta\eta$-equivalence, we thus need to establish that the normalization function satisfies the following two properties:

1. Normalization maps convertible terms to identical normal forms (we call it *soundness*: an equality in the syntax is also true in the semantics):

   ```
   soundness : {t u : Tm Γ σ} →
     t βη-≡ u → nf t ≡ nf u
   ```

2. Terms are convertible to their normal forms (we call it *completeness*, since it ensures that an equality in the semantics is also true in the syntax):

   ```
   completeness : (t : Tm Γ σ) → ⌈ nf t ⌉ βη-≡ t
   ```

A consequence of these two properties is that convertibility is exactly reflected by having the same normal forms:

```
{t u : Tm Γ σ} → t βη-≡ u ↔ nf t ≡ nf u
```

Since the equality of normal forms is obviously decidable (by simple inductions on types, contexts and normal forms), it follows that **convertibility is decidable**.

3.2 Auxiliary functions

We quickly present some auxiliary functions on normal forms and spines.

First, as for indices (see Section 2.2.3) and terms (see Section 2.3), we often need to weaken the context parameterizing a normal form or a spine. We hence have two functions to perform this weakening, whose definitions are direct adaptations of wkTm (so we do not give them here):

```
mutual
  wkNf : (x : Var Γ σ) → Nf (Γ - x) τ → Nf Γ τ
  wkSp : (x : Var Γ σ) → Sp (Γ - x) τ ρ → Sp Γ τ ρ
```

Spines are lists of normal forms; so while it is immediate to add a normal form at the *beginning* of a spine, we have to define a function that adds an element at the *end* of a spine:

```
appSp : Sp Γ ρ (σ ⇒ τ) → Nf Γ σ → Sp Γ ρ τ
appSp ε u = (u, ε)
appSp (t, ts) u = (t, appSp ts u)
```

3.3 The normalization function

We now define a *normalizer* as an Agda function that transforms a term into a normal form of the same type. This normalizer implements *hereditary substitutions* [17]. The idea behind these substitutions is to perform a syntactical substitution and normalize a term at the same time.

The normalization function is defined with the aid of two auxiliary functions that perform η-expansion, and four auxiliary functions that perform β-reduction.

3.3.1 η-expansion

Variables and neutral terms are η-expanded as much as possible depending on their types:

```
mutual
  nvar : forall {σ} → Var Γ σ → Nf Γ σ
  nvar {σ} x = ne2nf {σ} (x, ε)
  ne2nf : forall {σ} → Ne Γ σ → Nf Γ σ
  ne2nf {○} xns = ne xns
  ne2nf {σ ⇒ τ} (x, ns) =
    λn (ne2nf {τ} (vs x, appSp (wkSp vz ns) (nvar {σ} vz)))
```

3.3.2 β-reduction

Four functions perform the β-reduction:

- The function `_ [ _ := _ ]` substitutes a variable by a normal form inside a normal form.
- The function `_ < _ := _ >` substitutes a variable by a normal form inside a spine.
- The function `_ ◇ _` applies a normal form to a spine.
- The function `napp` launches the β-reduction.

```
mutual
  _ [ _ := _ ] : Nf Γ τ → (x : Var Γ σ) →
    Nf (Γ - x) σ → Nf (Γ - x) τ
  (λn t) [ x := u ] = λn (t [(vs x) := (wkNf vz u)])
  (ne (y, ts)) [ x := u ] with eq x y
  (ne (x, ts)) [ .x := u ] | same = u ◇ (ts < x := u >)
  (ne (. (wkv x y'), ts)) [ .x := u ] | diff x y' =
    ne (y', ts < x := u >)

_ < _ := _ > : Sp Γ τ ρ → (x : Var Γ σ) →
  Nf (Γ - x) σ → Sp (Γ - x) τ ρ
ε < x := u > = ε
(t, ts) < x := u > = (t [ x := u ]), (ts < x := u >)

_ ◇ _ : Nf Γ σ → Sp Γ σ τ → Nf Γ τ
t ◇ (u, us) = (napp t u) ◇ us
t ◇ ε = t

napp : Nf Γ (σ ⇒ τ) → Nf Γ σ → Nf Γ τ
napp (λn t) u = t [ vz := u ]
```

We now describe the mechanism behind these functions.
`t [x := u]`:

1. syntactically substitutes `x` by `u` in `t`; and
2. normalizes the result

at the same time: it is the essence of hereditary substitutions. Its definition is a case analysis on `t`:

- If `t` is `λn t'`, then substitute `x` by `u` in `t'` (`x` and `u` have to be weakened to avoid capture).
- If `t` is `ne (x, ts)`, then *apply* `u` to `ts'`, where `ts'` is `ts` where `x` is substituted by `u`. This application is the role of the `_ ◇ _` function.
- If `t` is `ne (y, ts)` where `y` does not represent the same variable as `x`, then `y` is unchanged and the substitution carries on in `ts`.

`ts < x := u >` substitutes `x` by `u` in all the normal forms appearing in `ts`. It is a simple case analysis on `ts`.

`t ◇ ts` successively applies `t` to each element of `ts`, by a simple case analysis on `ts`.

`napp t u` β-reduces the application of a normal form `t` to a normal form `u`, by a call to `_ [ _ := _ ]`. As `t` has a functional type, the typing ensures that it is a λ-abstraction.

3.3.3 The normalization function

The definition of the normalization function is now very straightforward: variables are η-expanded, applications are β-reduced, and λs are normalized under the λ:

```
nf : Tm Γ σ → Nf Γ σ
nf (var x) = nvar x
nf (Λ t)   = λn (nf t)
nf (app t u) = napp (nf t) (nf u)
```

We notice that Agda's type system ensures that normalization is **type preserving**: the type of the output normal form is the same as the type of the input term.

3.4 Termination of the normalizer

For soundness reasons, Agda is a total language: only terminating programs can be written. It automatically checks termination using a variant of the termination checker of `foetus` [4]: it computes the *completed call graph* of the functions and the corresponding *call matrices*. This process can only search for *structural* arguments to establish termination. One major interest of hereditary substitutions is to be structurally recursive, which is recognized by Agda.

The first two functions `nvar` and `ne2nf` are decreasing on the type of their arguments; their termination is thus very simple to establish.

We detail how Agda's termination checker establishes termination for the four functions `_ [ _ := _ ]`, `_ < _ := _ >`, `_ ◇ _` and `napp`. For an obvious matter of intelligibility, we are not going to give all the call matrices[1], but only the call graph with the relevant call matrices. For a different formalization, [3] gives a proof of the termination of hereditary substitutions using sized types.

Table 1 associates a lexicographical combination of structural orders to each function. We have written the types that are involved in the termination argument *à la* Church to make things clearer. **Figure 1** represents the call graph of the four functions:

- Each node corresponds to a function. We recall the lexicographical orders described in **Table 1** under the name of each function.
- Each edge corresponds to a possible call of a function by another function. It is labelled with a summary of the corresponding call matrix (when a function calls itself, we add a prime symbol to the call arguments). In any circuit in the graph with starting and ending vertex s, the measure associated to s decreases for the lexicographical order. It ensures that these mutual definitions terminate.

[1] The call matrices can be obtained by compiling our source file `hsubst.agda` with the option `-v 5`, for instance.

Function	$t_1\ [x_1\ ^{\sigma_1} := u_1]$	$ts_2 < x_2\ ^{\sigma_2} := u_2 >$
Measure	(σ_1, t_1)	(σ_2, ts_2)

Function	$t_3\ ^{\sigma_3} \diamond ts_3$	$napp\ t_4\ ^{\sigma_4}\ u_4$
Measure	(σ_3)	(σ_4)

Table 1. Decreasing measures in hereditary substitutions

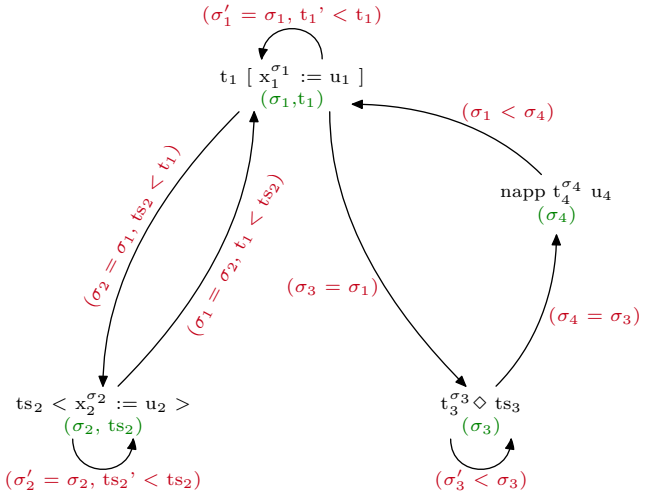

Figure 1. Call-graph in hereditary substitutions

3.5 Analyzing the normalizer

Once the definitions are established, the remainder of our development consists of an analysis of the normalizer in order to prove completeness and soundness (see Section 3.1). The next two sections give some hints how to perform these proofs, with few details. One can consult the source code [2] for a higher level of detail.

In our context, the proofs in Agda mainly rely on two techniques, that we call *generalization* and the introduction of *commutation lemmas*. Only one proof presented in Section 5 needs special care.

3.5.1 Generalization

As all the functions are inductively defined, it is natural to conduct the proofs by induction as well. As a consequence, in order to prove a statement, we very often need to *generalize* over some variables appearing in it; otherwise, some of the induction cases are not provable.

For instance, we want to prove

```
congSubst' : (t : Tm (Γ , σ) τ) → u₁ βη-≡ u₂ →
  subst t vz u₁ βη-≡ subst t vz u₂
```

by induction over t. The variable and application cases do not pose any problem, but in the abstraction case, we would have to prove:

```
subst t (vs vz) (wkTm vz u₁) βη-≡
  subst t (vs vz) (wkTm vz u₂)
```

which does not ensue from the induction hypothesis. To make it work, we have to generalize over vz into:

```
congSubst : (t : Tm Γ τ) → (x : Var Γ σ) →
  u₁ βη-≡ u₂ → subst t x u₁ βη-≡ subst t x u₂
```

3.5.2 Commutation lemmas

Most of the intermediary results we introduce are *commutation lemmas*. Informally, these are statements of the form `f (g x) ∼ g (f x)` where ∼ is either $\beta\eta$-≡ or ≡. For instance, in order to prove completeness (Section 4), we introduce the following lemma:

```
compApp : (t₁ : Nf Γ (σ ⇒ τ)) → (t₂ : Nf Γ σ) →
  ⌈ napp t₁ t₂ ⌉ βη-≡ app ⌈ t₁ ⌉ ⌈ t₂ ⌉
```

that states that embedding (⌈_⌉) and application (napp and app) commute.

4. Proof of completeness

Completeness states that any term is $\beta\eta$-equivalent to its normal form. To show this property, we have to establish it for all the auxiliary functions that define nf in Section 3.3: all these functions have to return a term that is $\beta\eta$-equivalent to their argument.

For the two functions that perform η-expansion, we have to prove the following properties:

Lemma 1. *The η-expansion of a term t is $\beta\eta$-equivalent to t:*

```
mutual
  compNe : (n : Ne Γ σ) → ⌈ ne2nf n ⌉ βη-≡ embNe n
  compVar : (v : Var Γ σ) → ⌈ nvar v ⌉ βη-≡ var v
```

Proof. The two properties are proved by a mutual induction:

- over σ for compNe;
- using compNe for compVar. □

For the four functions that perform β-reduction, we have to prove commutation lemmas between the functions that define hereditary substitutions (_ [_:=_], _<_:=_>, ◇ and napp) and the functions that define the embedding from normal forms to terms (⌈_⌉ and embSp):

Lemma 2. *Hereditary substitutions and embeddings commute:*

```
mutual
  substEmbSp : (ts : Sp Γ τ ρ) → (x : Var Γ σ) →
    (t : Nf (Γ - x) σ) → (acc : Tm Γ τ) →
    embSp (ts < x := t >) (subst acc x ⌈ t ⌉) βη-≡
      subst (embSp ts acc) x ⌈ t ⌉

appNfEmbSp : (u : Nf Γ σ) → (ts : Sp Γ σ ∘) →
  ⌈ u ◇ ts ⌉ βη-≡ embSp ts ⌈ u ⌉

substNfSubst : (t : Nf Γ τ) → (x : Var Γ σ) →
  (u : Nf (Γ - x) σ) →
  ⌈ t [x := u] ⌉ βη-≡ subst ⌈ t ⌉ x ⌈ u ⌉

compApp : (t₁ : Nf Γ (σ ⇒ τ)) → (t₂ : Nf Γ σ) →
  ⌈ napp t₁ t₂ ⌉ βη-≡ app ⌈ t₁ ⌉ ⌈ t₂ ⌉
```

Proof. The four properties are proven by a mutual induction:

- over ts for substEmbSp;
- over ts for appNfEmbSp;
- over t for substNfSubst;
- over t_1 for compApp. □

We are now able to establish our main theorem:

Theorem 1 (Completeness)**.** *Terms are convertible to their normal forms:*

```
completeness : (t : Tm Γ σ) → ⌈ nf t ⌉ βη-≡ t
```

Proof. By induction over t using compVar (**Lemma 1**) and compApp (**Lemma 2**). □

5. Proof of soundness

Soundness states that the normalization function identifies two $\beta\eta$-equivalent terms. We are going to prove it by induction over the proof of $\beta\eta$-equivalence of the two terms.

This proof of soundness is longer than the proof of completeness, which is not really surprising since we have to provide a proof of *equality*, which is a relation more restrictive than the $\beta\eta$-*equivalence*. But we also get stuck on the generalization of one lemma, that makes us introduce a new predicate to be able to formulate it. We now explain the intuition about the reasons of the introduction of this predicate.

In order to prove soundness in the case where t $\beta\eta$-$\equiv$ u is the rule eta, we would like to show that, if u is a normal form, then the η-expansion of u is equal to u:

```
etaEq : (u : Nf Γ (σ ⇒ τ)) →
  λn (napp (wkNf vz u) (nvar vz)) ≡ u
```

As u is a normal form which has a functional type, it is a λ-abstraction λn t for a certain normal form t. So this lemma is equivalent to proving the following proposition:

Proposition 1.

```
(t : Nf Γ τ) → wkNf (vs vz) t [vz := nvar vz] ≡ t
```

As explained in Section 3.5.1, this proof must be performed by induction over t; and to be able to conclude in the inductive case where t is an abstraction, we have to generalize over vz.

However, here, it is not as simple as in the example above: we are possibly misled by De Bruijn indices once more. We recall that the type of the function _ [x := u] (described in section 3.3) states that if x is typed in a context Γ, then u is typed in Γ - x. It means that in **Proposition 1**, **the two occurrences of vz do not represent the same variable**. In fact, they represent **two consecutive variables** in one context (which have the same type).

Hence, to generalize **Proposition 1**, it is necessary to generalize the two occurrences of vz with two different names. But it is important to take into account the fact that they are consecutive, otherwise the lemma would not be correct. This is why we introduce a new predicate onediff that precisely identifies consecutive indices in one context.

5.1 The predicate onediff

This predicate is very simple to define by induction: vz and vs vz follow one another; and if x and y follow one another, then vs x and vs y too.

```
data onediff : Var Γ σ → Var Γ τ → Set where
  odz : onediff vz (vs vz)
  ods : (x : Var Γ σ) → (y : Var Γ τ) → onediff x y
    → onediff (vs x) (vs y)
```

It is important to notice that onediff satisfies the following property:

Lemma 3. *If j and i follow one another in* Γ*, then* Γ - *i and* Γ - *j are equal:*

```
onediffMinus : (i j : Var Γ σ) → onediff j i →
  Γ - i ≡ Γ - j
```

Proof. By induction over onediff j i. □

since it allows us to transform a variable, a term or a normal form u typed in the context Γ - i into the same object typed in Γ - j for some Γ, i and j.

If p : Γ $\equiv$ Δ and u is parameterized by Γ, then ! p > u is u parameterized by Δ. So, in the example above, if p: onediff j i, then the result of the transformation is ! onediffMinus i j p > u.

5.2 The η-equality for normal forms

The predicate defined in the previous section now allows us to generalize **Proposition 1** that way (detailed above):

Lemma 4.

```
substNfEq : (i : Var Γ τ) → (t : Nf (Γ - i) σ) →
  (j : Var Γ τ) → (k : Var (Γ - j) τ) → (p : onediff j i)
  → wkv i (! sym (onediffMinus i j p) > k) ≡ j →
  (wkNf i t) [j := (nvar k)] ≡ ! onediffMinus i j p > t
```

The intuition behind this statement is that we generalize in **Proposition 1** all the indices that appear:

```
(wkNf i t) [j := (nvar k)]
```

and add the two constraints:

- onediff j i: j and i are two consecutive indices;
- wkv i (! sym (onediffMinus i j p) > k) ≡ j: j and wkv i (! sym (onediffMinus i j p) > k) represent the same variable.

These two conditions are sufficient to establish **Lemma 4**, and are verified by **Proposition 1**. Its proof only requires techniques presented in Section 3.5.

The η-equality for normal forms is now a direct consequence of substNfEq:

Lemma 5. *The η-equality is true for normal forms:*

```
etaEq : (u : Nf Γ (σ ⇒ τ)) →
  λn (napp (wkNf vz u) (nvar vz)) ≡ u
etaEq (λn u) =
  reflλn (substNfEq (vs vz) u vz vz odz refl)
```

5.3 The β-equality for normal forms

Similarly to completeness, to prove soundness in the case where $\beta\eta$-$\equiv$ is beta, we have to prove the following commutation lemma:

Lemma 6. *Normalization and substitution commute:*

```
nfSubstNf : (t : Tm Γ τ) → (x : Var Γ σ) →
  (u : Tm (Γ - x) σ) → (nf t) [x := (nf u)] ≡
    nf (subst t x u)
```

The proof is a rather technical but uses techniques presented in Section 3.5.

5.4 Proof of soundness

We are now able to establish our main theorem:

Theorem 2 (Soundness)**.** *The normal forms of two convertible terms are equal:*

```
soundness : {t u : Tm Γ σ} → t βη-≡ u →
  nf t ≡ nf u
```

Proof. By induction over t $\beta\eta$-$\equiv$ u, using etaEq (**Lemma 5**) in the eta case and and nfSubstNf (**Lemma 6**) in the beta case. The other cases are trivial. □

Conclusion of Sections 4 and 5: The reverse of **Theorem 2** (soundness) is a direct consequence of **Theorem 1** (completeness):

Consequence 1. *Two terms whose normal forms are equal are convertible.*

$$\mathsf{convertnf} : (\mathsf{t}\ \mathsf{u} : \mathsf{Tm}\ \Gamma\ \sigma) \to \mathsf{nf}\ \mathsf{t} \equiv \mathsf{nf}\ \mathsf{u} \to \mathsf{t}\ \beta\eta\text{-}\equiv\ \mathsf{u}$$

It follows that two terms are $\beta\eta$-equivalent if and only if their normal forms are equal. As equality on normal forms is obviously decidable, we can conclude that **$\beta\eta$-equivalence is decidable**.

Moreover, we expect that our algorithm that decides equivalence (normalize and check the equality of normal forms) uses only first order primitive recursion. We did not prove it formally, but it seems reasonable to say that it follows from construction. It makes clear that **$\beta\eta$-equivalence is primitive recursive**.

6. Related work

Hereditary substitutions were first introduced by Watkins et al. [17] to define a normalizer for the Concurrent Logical Framework. Their property to preserve canonical forms during substitution makes them a nice approach to an implementation of substitutions for Logical Frameworks [9, 12] and Higher-Order Abstract Syntax [15]. Abel [3] already noticed that the fact that hereditary substitutions are structurally recursive makes it easy to automatically check the termination of the algorithm they provide. We exploit the two properties:

- The fact that canonical forms are preserved by hereditary substitutions ensures the correct typing given to the substitution function.
- The fact that it is structurally recursive allows us to implement it in Agda without any need of an explicit termination proof.

The lemmas presented in this paper do not bring something new by their statements, but by their proofs. In fact, hereditary substitutions were studied in many different contexts [5, 12, 14], and most of our theorems are already proved in those papers. The main difference is that they use a named representation of variables.

Indeed, one major contribution of this paper is to adapt hereditary substitutions to De Bruijn indices (with the drawbacks we know, but with advantage that due to the fact that closed terms have a unique representation) and to implement it in an interactive theorem prover based on Type Theory.

Using **dependant types** in order to define directly the set of well-typed terms is not new. Its formalization has been studied in different interactive theorem provers based on dependant types, such as Agda [7] and Coq [8].

Also related is David's work on arithmetical **proofs of normalization results**, e.g. see [10]. Besides, [6] uses big step normalization to show decidability of a substitution calculus - this work has also been formalized in Agda.

7. Conclusion

This paper implements in Agda a normalizer for the simply typed λ-calculus, and proves that this normalizer can be used to decide the $\beta\eta$-equality over terms. It exploits some aspects of Agda and, more generally, of programming languages with dependent types:

- Dependent types are not only used to express theorems' statements, but also serve to define particular sets. Here, the parameterization of variables, terms and normal forms by contexts and types is a useful and natural way to define only the subset of λ-terms we are interested in. It is important to notice that Agda's implementation of dependent types makes it straightforward to manipulate such objects. Agda's dependent typing also ensures properties about functions just "by definition".
- Agda's termination checker is powerful enough to analyze non trivial mutually inductive definitions.
- Conversely, hereditary substitutions give a nice way to define a normalizer within total Type Theory, without constructing a proof of termination of a partial function. This could be extended to more complex type systems, for instance with polymorphism [3].

Acknowledgments

We thank Conor McBride, who suggested the idea and introduced some very convenient notations such as Γ - x. We also thank Andreas Abel for his useful help and comments about hereditary substitutions. We are grateful to the anonymous reviewers for their accurate comments and suggestions.

References

[1] Agda wiki. http://wiki.portal.chalmers.se/agda.

[2] The decidability of the $\beta\eta$-equivalence using hereditary substitutions. http://www.lix.polytechnique.fr/~keller/Recherche/hsubst.html.

[3] A. Abel. Implementing a normalizer using sized heterogeneous types. *Workshop on Mathematically Structured Functional Programming, MSFP*, 2006.

[4] A. Abel and T. Altenkirch. A predicative analysis of structural recursion. *J. Funct. Program.*, 12(1):1–41, 2002.

[5] A. Abel and D. Rodriguez. Syntactic Metatheory of Higher-Order Subtyping. In M. Kaminski and S. Martini, editors, *CSL*, volume 5213 of *Lecture Notes in Computer Science*, pages 446–460. Springer, 2008. ISBN 978-3-540-87530-7.

[6] T. Altenkirch and J. Chapman. Tait in one big step. In *Workshop on Mathematically Structured Functional Programming, MSFP*, volume 2006. Citeseer, 2006.

[7] T. Altenkirch and B. Reus. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In J. Flum and M. Rodríguez-Artalejo, editors, *CSL*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. ISBN 3-540-66536-6.

[8] N. Benton, A. Kennedy, and C. Hur. Strongly typed term representations in Coq.

[9] K. Crary. Explicit Contexts in LF (Extended Abstract). *Electr. Notes Theor. Comput. Sci.*, 228:53–68, 2009.

[10] R. David. Normalization without reducibility. *Annals of pure and applied logic*, 107(1-3):121–130, 2001.

[11] N. G. De Bruijn. Lambda Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation With Application to the Church-Rosser Theorem. *Indag. Math.*, pages 381–382, 1972.

[12] R. Harper and D. R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.

[13] U. Norell. Towards a Practical Programming Language Based on Dependent Type Theory. *PhD thesis, Chalmers Univ. of Tech.*, 2007.

[14] F. Pfenning. Church and Curry: Combining intrinsic and extrinsic typing. *Studies in Logic and the Foundations of Mathematics*, 2008.

[15] B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In G. C. Necula and P. Wadler, editors, *POPL*, pages 371–382. ACM, 2008. ISBN 978-1-59593-689-9.

[16] B. Pierce. *Types and programming languages*. The MIT Press, 2002. Chapter 12: Normalization.

[17] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A Concurrent Logical Framework: The Propositional Fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2003. ISBN 3-540-22164-6.