

Normalization by hereditary substitutions

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 [], the most computational one is to give a normalization function which 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` [11, 1]. `Agda` is a programming language with dependent types. We exploit this property both:

- to define functions whose some properties are ensured

^{*}Secondary affiliation: École Normale Supérieure de Lyon, France

by their types;

- to use `Agda` as an interactive theorem prover.

The source code can be found online [2]. We thus use `Agda` as our metalanguage all along the paper.

The main particularities of our implementation are:

- 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) [13], which preserve **canonical forms** by substituting and reducing the redices that can appear from substitution at the same time. An important aspect of hereditary substitutions is that they can be defined in a structurally manner; hence, they 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 works (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 [6], 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
  o : 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 infix operators.

For terms, we use (typed) De Bruijn indices to represent variables [9]. 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
  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 inferred 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, and to be suitable for a machine. It is well-know that it is not suitable for a human being, though. For clarity, this section gives 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 $x : \text{Var } \Gamma \sigma$, we call x an *index*.
- If $x : \text{Var } \Gamma \sigma$, 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 -term¹:

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

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 often use 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 **Eq** specifying if two indices parameterized by the same context represent the same variable or not. Its definition 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 $x \equiv \text{wkv } y \ z$ (\equiv stands for Leibniz equality, with the three usual functions **refl**, **sym** and **trans**).

The intuition behind the second property is that if x and y do not represent the same variable, then x “exists” in $\Gamma - y$:

¹We use Krivine’s notation to write λ -terms. In applications, brackets surround the function; for instance: $(f) x$. $(f) x y$ is a shortcut for $((f) x) y$.

this is the index z . x is thus the weakening of z when y is added to the context.

As a consequence, the predicate of equality 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)

```

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 v in some term t by another term. The type we give to the substitution function is: $(t : Tm \Gamma \tau) \rightarrow (x : Var \Gamma \sigma) \rightarrow (u : Tm (\Gamma - x) \sigma) \rightarrow Tm (\Gamma - x) \tau$. 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 $\Gamma - x$, 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 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 types 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 $([_])$:

mutual

$$\begin{aligned} \llbracket _ \rrbracket &: \text{Nf } \Gamma \sigma \rightarrow \text{Tm } \Gamma \sigma \\ \llbracket \lambda n \ n \rrbracket &= \Lambda \llbracket n \rrbracket \\ \llbracket ne \ n \rrbracket &= \text{embNe } n \\ \text{embNe} &: \text{Ne } \Gamma \sigma \rightarrow \text{Tm } \Gamma \sigma \\ \text{embNe } (v, s) &= \text{embSp } s \ (\text{var } v) \\ \text{embSp} &: \text{Sp } \Gamma \sigma \tau \rightarrow \text{Tm } \Gamma \sigma \rightarrow \text{Tm } \Gamma \tau \\ \text{embSp } \varepsilon \ \text{acc} &= \text{acc} \\ \text{embSp } (n, s) \ \text{acc} &= \text{embSp } s \ (\text{app } \text{acc } \llbracket n \rrbracket) \end{aligned}$$

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` : $\text{Tm } \Gamma \sigma \rightarrow \text{Nf } \Gamma \sigma$ that should satisfy the following properties:

1. Terms are convertible to their normal forms (we call it *completeness*):

$$\text{completeness} : (t : \text{Tm } \Gamma \sigma) \rightarrow \llbracket \text{nf } t \rrbracket \beta\eta \equiv t$$

2. Normalization maps convertible terms to identical normal forms (we call it *soundness*):

$$\text{soundness} : \{t \ u : \text{Tm } \Gamma \sigma\} \rightarrow t \beta\eta \equiv u \rightarrow \text{nf } t \equiv \text{nf } u$$

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

$$\{t \ u : \text{Tm } \Gamma \sigma\} \rightarrow t \beta\eta \equiv u \leftrightarrow \text{nf } t \equiv \text{nf } u$$

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 under which a normal form or a spine is typed. We hence have two functions to perform this weakening, whose definitions are direct adaptations of `wkTm` (so we do not give them here):

mutual

$$\begin{aligned} \text{wkNf} &: (x : \text{Var } \Gamma \sigma) \rightarrow \text{Nf } (\Gamma - x) \tau \rightarrow \text{Nf } \Gamma \tau \\ \text{wkSp} &: (x : \text{Var } \Gamma \sigma) \rightarrow \text{Sp } (\Gamma - x) \tau \rho \rightarrow \text{Sp } \Gamma \tau \rho \end{aligned}$$

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:

$$\begin{aligned} \text{appSp} &: \text{Sp } \Gamma \rho \ (\sigma \Rightarrow \tau) \rightarrow \text{Nf } \Gamma \sigma \rightarrow \text{Sp } \Gamma \rho \tau \\ \text{appSp } \varepsilon \ u &= (u, \varepsilon) \\ \text{appSp } (t, \text{ts}) \ u &= (t, \text{appSp } \text{ts } u) \end{aligned}$$

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* [13]. 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

$$\begin{aligned} \text{nvar} &: \text{Var } \Gamma \sigma \rightarrow \text{Nf } \Gamma \sigma \\ \text{nvar } x &= \text{ne2nf } (x, \varepsilon) \\ \text{ne2nf} &: \text{forall } \{\sigma\} \rightarrow \text{Ne } \Gamma \sigma \rightarrow \text{Nf } \Gamma \sigma \\ \text{ne2nf } \{o\} \ \text{xns} &= \text{ne } \text{xns} \\ \text{ne2nf } \{\sigma \Rightarrow \tau\} \ (x, \text{ns}) &= \\ &\lambda n \ (\text{ne2nf } (vs \ x, \text{appSp } (\text{wkSp } \text{vz } \text{ns}) \ (\text{nvar } \text{vz}))) \end{aligned}$$

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 `_ \diamond _` applies a normal form to a spine.
- The function `napp` launches the β -reduction.

mutual

$$\begin{aligned} _ [:=] &: \text{Nf } \Gamma \tau \rightarrow (x : \text{Var } \Gamma \sigma) \rightarrow \\ &\text{Nf } (\Gamma - x) \sigma \rightarrow \text{Nf } (\Gamma - x) \tau \\ (\lambda n \ t) [x := u] &= \lambda n \ (t \ [(vs \ x) := (\text{wkNf } \text{vz } u)]) \\ (\text{ne } (y, \text{ts})) [x := u] \ \text{with } \text{eq } x \ y & \\ (\text{ne } (x, \text{ts})) [x := u] \ \mid \ \text{same} &= u \ \diamond \ (\text{ts} < x := u >) \\ (\text{ne } (. (\text{wkv } x \ y'), \text{ts})) [x := u] \ \mid \ \text{diff } x \ y' &= \\ &\text{ne } (y', \text{ts} < x := u >) \end{aligned}$$
$$_ < := > : \text{Sp } \Gamma \tau \rho \rightarrow (x : \text{Var } \Gamma \sigma) \rightarrow \\ \text{Nf } (\Gamma - x) \sigma \rightarrow \text{Sp } (\Gamma - x) \tau \rho$$
$$\varepsilon < x := u > = \varepsilon$$
$$(t, \text{ts}) < x := u > = (t [x := u]), (\text{ts} < x := u >)$$
$$_ \diamond _ : \text{Nf } \Gamma \sigma \rightarrow \text{Sp } \Gamma \sigma \tau \rightarrow \text{Nf } \Gamma \tau$$
$$t \diamond (u, \text{us}) = (\text{napp } t \ u) \ \diamond \ \text{us}$$
$$t \diamond \varepsilon = t$$
$$\text{napp} : \text{Nf } \Gamma (\sigma \Rightarrow \tau) \rightarrow \text{Nf } \Gamma \sigma \rightarrow \text{Nf } \Gamma \tau$$
$$\text{napp } (\lambda n \ t) \ u = t \ [\text{vz} := u]$$

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 $\lambda n t'$, 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 $_ \diamond _$ 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 .

$t \diamond ts$ 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 λ s are normalized under the λ :

$$\begin{aligned} nf &: \text{Tm } \Gamma \sigma \rightarrow \text{Nf } \Gamma \sigma \\ nf(\text{var } x) &= \text{nvar } x \\ nf(\Lambda t) &= \lambda n (nf t) \\ nf(\text{app } t u) &= \text{napp } (nf t) (nf u) \end{aligned}$$

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** [3]: 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 $_ [:= _]$, $_ < := _ >$, $_ \diamond _$ and **napp**. For an obvious matter of intelligibility, we are not going to give all the call matrices², but only the call graph with

²The 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

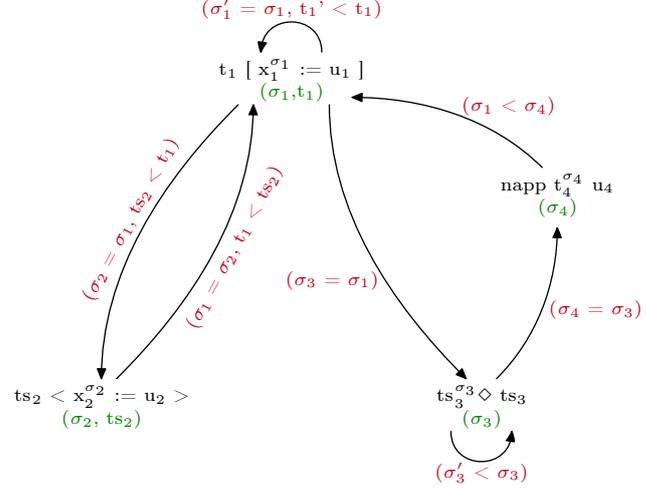


Figure 1: Call-graph in hereditary substitutions

the relevant call matrices. For a different formalization, [4] gives a proof of the termination of hereditary substitutions using sized typed.

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.

3.5 Analyzing the normalizer

Once the definitions established, the remaining of our development consists in 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 report to the source code for a higher level of details.

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

$$\text{congSubst}' : (t : \text{Tm } (\Gamma, \sigma) \tau) \rightarrow u_1 \beta\eta\equiv u_2 \rightarrow \\ \text{subst } t \text{ vz } u_1 \beta\eta\equiv \text{subst } t \text{ vz } u_2$$

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

$$\text{subst } t \text{ (vs vz)} (\text{wkTm } \text{vz } u_1) \beta\eta\equiv \\ \text{subst } t \text{ (vs vz)} (\text{wkTm } \text{vz } u_2)$$

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

$$\text{congSubst} : (t : \text{Tm } \Gamma \tau) \rightarrow (x : \text{Var } \Gamma \sigma) \rightarrow \\ u_1 \beta\eta\equiv u_2 \rightarrow \text{subst } t \times u_1 \beta\eta\equiv \text{subst } t \times u_2$$

3.5.2 Commutation lemmas

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

$$\text{compApp} : (t_1 : \text{Nf } \Gamma (\sigma \Rightarrow \tau)) \rightarrow (t_2 : \text{Nf } \Gamma \sigma) \rightarrow \\ \lceil \text{napp } t_1 \ t_2 \rceil \beta\eta\equiv \text{app } \lceil t_1 \rceil \lceil t_2 \rceil$$

that states that embedding ($\lceil _ \rceil$) 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 :*

$$\text{mutual} \\ \text{compNe} : (n : \text{Ne } \Gamma \sigma) \rightarrow \lceil \text{ne2nf } n \rceil \beta\eta\equiv \text{embNe } n \\ \text{compVar} : (v : \text{Var } \Gamma \sigma) \rightarrow \lceil \text{nvar } v \rceil \beta\eta\equiv \text{var } v$$

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 ($_ \lceil _ \rceil$, $_ \langle _ \rceil$, \diamond and napp) and the functions that define the embedding from normal forms to terms ($\lceil _ \rceil$ and embSp):

Lemma 2 *Hereditary substitutions and embeddings commute:*

$$\text{mutual} \\ \text{substEmbSp} : (ts : \text{Sp } \Gamma \tau \rho) \rightarrow (x : \text{Var } \Gamma \sigma) \rightarrow \\ (t : \text{Nf } (\Gamma - x) \sigma) \rightarrow (\text{acc} : \text{Tm } \Gamma \tau) \rightarrow \\ \text{embSp } (ts \langle x := t \rangle) (\text{subst } \text{acc } x \lceil t \rceil) \beta\eta\equiv \\ \text{subst } (\text{embSp } ts \text{ acc}) \times \lceil t \rceil$$

$$\text{appNfEmbSp} : (u : \text{Nf } \Gamma \sigma) \rightarrow (ts : \text{Sp } \Gamma \sigma \circ) \rightarrow \\ \lceil u \diamond ts \rceil \beta\eta\equiv \text{embSp } ts \lceil u \rceil$$

$$\text{substNfSubst} : (t : \text{Nf } \Gamma \tau) \rightarrow (x : \text{Var } \Gamma \sigma) \rightarrow \\ (u : \text{Nf } (\Gamma - x) \sigma) \rightarrow \\ \lceil t \lceil x := u \rceil \rceil \beta\eta\equiv \text{subst } \lceil t \rceil \times \lceil u \rceil$$

$$\text{compApp} : (t_1 : \text{Nf } \Gamma (\sigma \Rightarrow \tau)) \rightarrow (t_2 : \text{Nf } \Gamma \sigma) \rightarrow \\ \lceil \text{napp } t_1 \ t_2 \rceil \beta\eta\equiv \text{app } \lceil t_1 \rceil \lceil t_2 \rceil$$

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:*

$$\text{completeness} : (t : \text{Tm } \Gamma \sigma) \rightarrow \lceil \text{nf } t \rceil \beta\eta\equiv t$$

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 are also stuck by 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 is 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 :

$$\text{etaEq} : (u : \text{Nf } \Gamma (\sigma \Rightarrow \tau)) \rightarrow \lambda n (\text{napp } (\text{wkNf } \text{vz } u) (\text{nvar } \text{vz})) \equiv u$$

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

Proposition 1

$$(t : \text{Nf } \Gamma \tau) \rightarrow \text{wkNf } (\text{vs } \text{vz}) t [\text{vz} := \text{nvar } \text{vz}] \equiv t$$

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 $\Gamma - x$. 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.

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 $\text{vs } \text{vz}$ follow one another; and if x and y follow one another, then $\text{vs } x$ and $\text{vs } y$ too.

$$\begin{aligned} \text{data onediff} & : \text{Var } \Gamma \sigma \rightarrow \text{Var } \Gamma \tau \rightarrow \text{Set where} \\ \text{odz} & : \text{onediff } \{(\Gamma, \sigma), \tau\} \text{vz } (\text{vs } \text{vz}) \\ \text{ods} & : (x : \text{Var } \Gamma \sigma) \rightarrow (y : \text{Var } \Gamma \tau) \rightarrow \text{onediff } x y \\ & \rightarrow \text{onediff } (\text{vs } x) (\text{vs } y) \end{aligned}$$

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

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

$$\text{onediffMinus} : (ij : \text{Var } \Gamma \sigma) \rightarrow \text{onediff } j i \rightarrow \Gamma - i \equiv \Gamma - j$$

PROOF. By induction over **onediff** $j i$. \square

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

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

5.2 The η -equality for normal forms

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

Lemma 4

$$\begin{aligned} \text{substNfEq} & : (i : \text{Var } \Gamma \tau) \rightarrow (t : \text{Nf } (\Gamma - i) \sigma) \rightarrow \\ & (j : \text{Var } \Gamma \tau) \rightarrow (k : \text{Var } (\Gamma - j) \tau) \rightarrow (p : \text{onediff } j i) \\ & \rightarrow \text{wkv } i (! \text{sym } (\text{onediffMinus } i j p) > k) \equiv j \rightarrow \\ & (\text{wkNf } i t) [j := (\text{nvar } k)] \equiv ! \text{onediffMinus } i j p > t \end{aligned}$$

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

$$(\text{wkNf } i t) [j := (\text{nvar } k)]$$

and add the two constraints:

- **onediff** $j i$: j and i are two consecutive indices;
- $\text{wkv } i (! \text{sym } (\text{onediffMinus } i j p) > k) \equiv j$: j and $\text{wkv } i (! \text{sym } (\text{onediffMinus } i j p) > k)$ 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 stands for normal forms:*

$$\begin{aligned} \text{etaEq} & : (u : \text{Nf } \Gamma (\sigma \Rightarrow \tau)) \rightarrow \\ & \lambda n (\text{napp } (\text{wkNf } \text{vz } u) (\text{nvar } \text{vz})) \equiv u \\ \text{etaEq} (\lambda n u) & = \\ \text{refl} \lambda n (\text{substNfEq } (\text{vs } \text{vz}) u \text{vz } \text{vz } \text{odz } \text{refl}) & \end{aligned}$$

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:*

$$\begin{aligned} \text{nfSubstNf} & : (t : \text{Tm } \Gamma \tau) \rightarrow (x : \text{Var } \Gamma \sigma) \rightarrow \\ & (u : \text{Tm } (\Gamma - x) \sigma) \rightarrow (\text{nf } t) [x := (\text{nf } u)] \equiv \\ & \text{nf } (\text{subst } t \times u) \end{aligned}$$

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:*

$$\text{soundness} : \{t\ u : \text{Tm } \Gamma\ \sigma\} \rightarrow t\ \beta\eta\text{-}\equiv\ u \rightarrow \\ \text{nf } t \equiv \text{nf } u$$

PROOF. By induction over $t\ \beta\eta\text{-}\equiv\ u$, using `etaEq` (Lemma 5) in the `eta` case and `nfSubstNf` (Lemma 6) in the `beta` case. The other cases are trivial. \square

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.*

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

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, our algorithm that decides equivalence (normalize and check the equality of normal forms) is primitive recursive, which makes clear that $\beta\eta$ -equivalence is primitive recursive.

6. RELATED WORK

Hereditary substitutions were first introduced by [13] et al. 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 [10, 7] and Higher-Order Abstract Syntax [12]. Abel [4] 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.

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.

Also related is David's work on arithmetical proofs of normalisation results, e.g. see [8]. [5] use big step normalisation 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 dependant 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 [4].

Acknowledgments. We thank Conor McBride, who suggested the idea and introduced some very convenient notations such as $\Gamma - x$. We also thank Andreas Abel for his useful help and comments about hereditary substitutions.

8. REFERENCES

- [1] `Agda` wiki. <http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php?n=Main.Documentation>.
- [2] The decidability of the $\beta\eta$ -equivalence using hereditary substitutions. <http://www.lix.polytechnique.fr/~keller/Recherche/hsubst.html>.
- [3] A. Abel. foetus-termination checker for simple functional programs. *Programming Lab Report*, 1998.
- [4] A. Abel. Implementing a normalizer using sized heterogeneous types. *Workshop on Mathematically Structured Functional Programming, MSFP*, 2006.
- [5] T. Altenkirch and J. Chapman. Tait in one big step. In *Workshop on Mathematically Structured Functional Programming, MSFP*, volume 2006. Citeseer, 2006.
- [6] 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.
- [7] K. Crary. Explicit Contexts in LF (Extended Abstract). *Electr. Notes Theor. Comput. Sci.*, 228:53–68, 2009.
- [8] R. David. Normalization without reducibility. *Annals of pure and applied logic*, 107(1-3):121–130, 2001.
- [9] 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.

- [10] R. Harper and D. R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.
- [11] U. Norell. Towards a Practical Programming Language Based on Dependent Type Theory. *PhD thesis, Chalmers Univ. of Tech.*, 2007.
- [12] 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.
- [13] 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.