LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - The University of Edinburgh

Brewing Strong Normalization Proofs with LEGO

by

Thorsten Altenkirch

LFCS Report Series ECS-LFCS-92-230

LFCS August 1992

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Copyright © 1992, LFCS

Copyright © 1992, Laboratory for Foundations of Computer Science
University of Edinburgh. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

Brewing Strong Normalization Proofs with LEGO

Thorsten Altenkirch
Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh

August 18, 1992

Abstract

This is the documentation of completely formalized strong normalization proofs for Church style presentations of simply typed λ Calculus and System F in LEGO. The underlying type theory is the Calculus of Constructions enriched by inductive types. The proofs follow Girard et al [GLT89] but we make essential use of general inductive types to simplify the presentation. I will use this example to discuss *Computer Aided Formal Reasoning* in general.

1 Introduction

I am going to describe a complete formalization of strong normalization proofs for simply typed λ calculus and System F in LEGO [LP92]. The proofs [1] use the tactics provided by the LEGO system. However, in the end we can extract a typed λ term which represents the proof. The proofs are complete, i.e. there are no non-logical axioms used. [2] This implies in particularly that the proof is intuitionistic, i.e. we never use excluded middle. As a consequence of this we can extract a normalization function together with its verification from the proof.

Strong normalization proofs are quite a fashionable subject for formal proofs. Stefano Berardi worked on a strong normalization proof for System F in Pure Constructions also using LEGO [Ber91]. Catarina Coquand [Coq92a] is working on a strong normalization proof for simply typed λ calculus using ALF [Mag92].

One reason why strong normalization proofs are interesting candidates for formalization is that they are fairly intricate and that they require a complete formalization and understanding of the calculus involved. Another reason is that everybody who works in the area of Type Theory or formal proofs knows them and studies them anyway.

[1] The complete lego code is not included in the report but can be obtained by ftp or email. See Appendix.

[2] To show the fourth Peano axiom $\forall_{n\in\mathcal{N}} n+1 \neq 0$ and its variants we need *large eliminations*. They have been investigated by Benjamin Werner recently ([Wer92]).

Usual notation	Lego notation	Remarks
$\lambda x : A.M$	`[x:A]M` *or* `[x|A]M`	If `[x|A]M` is used the argument is infered when the function is applied.
$\Pi_{x:A}B$	`{x:A}B` *or* `{x|A}M`	`{x|A}B` corresponds to `[x|A]M`.
MN	`M N` *or* `M|N`	`M|N` is used to supply an implicit argument.

Figure 1: Essential syntax of terms

2 Using LEGO

LEGO is a proof development system based on Type Theory which has been implemented by Randy Pollack. A good introduction to the use of LEGO can be found in [Hof92], where LEGO is used for program verification.

2.1 The Type Theory

The *standard* Type Theory used in LEGO is the Extended Calculus of Constructions (ECC) [Luo90]. However, here we do not exploit the extensions introduced by Luo: universes and strong Σ types — we only require one predicative universe on the top of the Pure Calculus of Constructions. We also diverge from Luo's proposal to use the predicative universes (`Type`) for computations and use the impredicative universe `Prop` for both: logic and computations . This is expressed by the definition `Set=Prop`. This means that philosophically we follow Martin-Löf's identification of propositions and types but we differ in that we accept impredicative quantification.

In earlier work on formal proofs in the Calculus of Constructions the so called *impredicative encodings* of inductive types have been used (e.g. see [Alt90]). They have a number of disadvantages, in particularly we have to assume an induction axiom to prove anything about them. This induction axiom is not interpreted computationally and destroys the computational interpretation of propositions because we now have non-canonical elements in every type. To overcome this problem we introduce inductive types explicitly and introduce elimination constants *à la* Martin-Löf together with typed reduction rules (e.g. [NPS90]). Our approach to inductive types is similar to the one chosen in Coq [D+91] and is based on [CP89].

2.2 The Logic

To understand the proof script it is not necessary (but helpful) to understand all LEGO commands. The structure of the proof should become clear from the sequence of lemmas and definitions.

The basic logical definitions are included by the command `Logic` — see figure 2. This uses an impredicative encoding of basic logical connectives. However, we define propositional equality `EQ` as an inductive type in the same way as in Martin-Löf Type Theory.

Usual notation	Lego notation	Definition
$A \wedge B$	`A /\ B` *or* `and A B`	`{C|Prop}(A->B->C) -> C`
$A \vee B$	`A \/ B` *or* `or A B`	`{C|Prop}(A->C) -> (B->C) -> C`
$A \Rightarrow B$	`A -> B`	`{_:A}B`
$\bot$ (falsity)	`absurd`	`{C:Prop} C`
$\neg A$	`not A`	`A -> absurd`
$A \Leftrightarrow B$	`(iff A B)`	`(and (A->B) (B->A))`
$\forall x \in A.P(x)$	`{x:A} (P x)` *or* `{x|A} (P x)`	*primitive*
$\exists x \in A.P(x)$	`Ex[a:A]P a` *or* `Ex P`	`{C|Prop}({a:A}(P a) -> C) -> C`
$x = y \in X$	`(EQ x y)`	*inductive type*

Figure 2: Logical connectives in Lego

Theoretically, it would have been better to use inductive types for all logical connectives because they come with stronger elimination rules and it seems a bit of a waste to introduce impredicativity just to encode basic logical connectives. However, the current `Refine` tactic of LEGO is tuned for the impredicative encodings.

2.3 Inductive types

So far inductive types have not been implemented in the LEGO system but there is the possibility of using typed rewriting rules to realize them. In the proof I use my own syntax for `mu` types and definitions by primitive recursion which is put as a comment [3] into the proof. This is then expanded into LEGO code - I will explain this by a simple example [4]:

```
mu[Nat:Set](zero:Nat,succ:Nat->Nat)
```

is the type of natural numbers. We translate this into LEGO by *introducing* the type and the canonical elements as constants:

```
$[Nat : Set]
$[zero : Nat]
$[succ : Nat->Nat]
```

To use natural numbers in proofs or computations we need an elimination constant which is introduced as well:

[3] LEGO comments are introduced by (* ... *).

[4] All the LEGO code discussed in this section can be found in `basic.l`

```
$[RecNat : {P:Nat->Type}
          (P zero)->({n:Nat}(P n)->(P (succ n)))
          -> {n:Nat}P n];
```

We also introduce the following typed rewriting rules which correspond to primitive recursion with natural numbers:

```
[[P:Nat->Type][z:P zero][f:{n:Nat}(P n)->(P (succ n))][n:Nat]
    RecNat P z f zero ==> z
 || RecNat P z f (succ n) ==> f n (RecNat P z f n)];
```

These definitions extend LEGO's definitional equality and we can perform computations with them. They also have the effect that no non-canonical elements are introduced because every occurrence of RecNat is eventually eliminated.

If we want to define a function over natural numbers we declare it first in an ML-like fashion:

```
add : Nat -> Nat -> Nat

rec add zero n = n
  | add (succ m) n = succ (add m n)
```

which can be (in a very mechanical way) translated into the following LEGO code which makes use of the recursor (here we use a derived non-dependent recursor RecNatN [5] to simplify the typing):

```
[RecNatN[C|Type] = RecNat ([_:Nat]C)];

[add = RecNatN ([n:Nat]n)
               ([m:Nat][add_m:Nat->Nat][n:Nat]succ (add_m n))];
```

This way we can only define functions by primitive recursion but note that we get more than the usual primitive recursive functions already because we have higher order functions.

It is interesting to consider inductive types with dependent constructors like the type of vectors :

```
[A:Set]mu[Vec:Nat->Set](v_nil:Vec zero,
                        v_cons:A->{n|Nat}(Vec n)->(Vec succ n))
```

Vectors resemble Lists but differ in that the length of the sequence is part of its type. Therefore we have Vec: Set -> Nat -> Set in contrast to List : Set -> Set, i.e. Vec A 3 [6] is the type of sequences of type A of length 3.

[5] We adopt the convention that $R\mathbb{N}$ stands for the non-dependent version of recursor R.

[6] Oops - I meant to say Vec A (succ (succ (succ zero))).

When we defined the elimination constant for Nat we allowed eliminations over an arbitrary universe — we call these *large eliminations.* We can use them to prove the fourth Peano axiom as in Martin-Löf Type Theory with universes (e.g. [Smi87]). However, it is interesting to observe that there is a purely computational use of *large eliminations*: We can apply them to realize the following *run-time-error-free* destructors for vectors:

```
[A|Set]

v_hd : {n|Nat}(Vec A (succ n))->A
rec v_hd|n (v_cons a _) = a;

v_tl : {n|Nat}(Vec A (succ n))->Vec A n
rec v_tl|n (v_cons _ l) = l;
```

Using these error free functions not only simplifies the verification of functions using vectors; it also opens a possibility, in principle, to compile code involving dependent types more efficiently.

Another use of dependent inductive types is the definition of predicates as the initial semantics of a set of Horn clauses [7]. An example is the definition of the predicate $\leq$ (LE) for natural numbers:

```
mu[LE:Nat->Nat->Set](
   le0:{n:Nat}LE zero n,
   le1:{m,n|Nat}(LE m n) -> (LE (succ m) (succ n)))
```

We also define the corresponding rewriting rules, which means that we can actually use proofs of LE for computations. An example would be the definition of the following dependently typed subtraction function:

```
sub : {m,n:Nat}(LE m n)->Nat

rec sub zero     m         (le0 n) = m
  | sub (succ m) (succ n) (le1 p) = succ (sub m n p)
```

which actually extracts a witness (the difference) from a proof of LE m n.

3 A guided tour through the LEGO code

In the following I am going to explain and comment the formalized proof. For more detailed information it may be worthwhile to obtain the complete LEGO code — see appendix.

[7] Actually it is a generalization of Horn clauses which we may call higher order Horn clauses.

3.1 A sketch of the proof

One of the essential insights about strong normalization proofs is that they require another form of induction than proofs of other properties of typed λ calculi like the *subject reduction property* or the *Church-Rosser property*. We cannot show strong normalization just by induction over type derivations or by induction over the length of a reduction. We have to apply another principle which may best be described as an *induction over the meaning of types.*

We are going to interpret every type (together with a context) by a set [8] of terms. We do this by assigning to every operator on types a semantic operator on sets of terms which is motivated by the elimination rule for this type — e.g. the interpretation of the arrow type $\sigma \to \tau$ is defined as the set of all terms that, if applied to a term in the interpretation of σ, are in the interpretation of τ. Now we have to show that the interpretation takes place in a family of sets of terms — here called the *Candidates of Reducibility*. This plays the role of an induction hypothesis: Every candidate set is a set of strongly normalizing terms but not the other way around. The idea of this is to achieve that indeed every type is interpreted as a candidate (we say it is a logical predicate) and also that every term is in the interpretation of its type (we say the interpretation is sound). Putting these things together implies that every typable term is strongly normalizing.

3.2 Overview over the LEGO code

The proofs are distributed over several files and some of them are shared between the proof for simply typed λ calculus and System F:

`load-simple.l` Includes all the files for simply typed λ calculus in the right sequence.

`load-f.l` Analogue for System F.

`basic.l` Definition of some basic inductive types (like `Nat`) which are not particular to the proofs here.

`lambda.l` Definition of untyped λ-terms using deBruijn indices. We also define substitution, weakening and reduction and prove a number of lemmas (in particular about substitution and weakening).

`snorm.l` We define strongly normalizing (`SN`) and Candidates of Reducibility (`CR`). All the essential lemmas of both proofs are done here (the rest is syntax).

`simple.l` Types and derivations for simply typed λ calculus are defined.

`sn-simple.l` We define an interpretation of judgements and show that it is logical and sound exploiting the lemmas proven in `snorm.l`. Strong normalization is derived as a simple corollary of this.

`f.l` Analogue to `simple.l`.

`sn-f.l` Analogue to `sn-simple.l`.

[8] The term *set* becomes a bit overloaded. Here we mean a set in the classical sense, i.e. the extension of a predicate and not the universe `Set`. A set of terms corresponds to the type `Tm -> Set`.

3.3 basic.l

To keep the proofs self contained we introduce all the basic types needed (and some trivial lemmas) here. Apart from Vec, which has been discussed before, we also introduce the family of finite types: Fin : Nat -> Set which is defined by:

```
mu[Fin:Nat->Set](f_zero:{n:Nat}Fin (succ n),
                 f_succ:{n|Nat}(Fin n)->(Fin (succ n)))
```

The idea behind this is that Fin i corresponds to a type with i canonical elements. We use Fin to define an error-free lookup function for vectors: [9]

```
v_nth : {n|Nat}(Fin n)->(Vec A n)->A

rec v_nth|(succ n) (f_zero n) = v_hd|A|n
  | v_nth|(succ n) (f_succ i) = o (v_tl|A|n) (v_nth i)
```

For the interpretation of System F we also need *large vectors*, i.e. vectors of type Nat->Type(0)->Type(0) (instead of Nat->Set->Set). Unfortunately, this sort of polymorphism can not be expressed in the current implementation of LEGO. Therefore we repeat all the definitions for large vectors (VEC).

3.4 lambda.l

We define untyped λ-terms (Tm) using de Bruijn indices [dB72] as the following inductive type:

```
mu[Tm:Set](var : Nat -> Tm,
           app : Tm -> Tm -> Tm,
           lam : Tm -> Tm)
```

It would have been possible to use a neater definition which makes the number of free variables explicit:

```
mu[Tm:Nat->Set](var : {n|Nat}(Fin n) -> (Tm n),
                app : {n|Nat}(Tm n) -> (Tm n) -> (Tm n),
                lam : {n|Nat}(Tm (succ n)) -> (Tm n) )
```

However, this makes the operations of weakening and substitution more difficult to define and it turns out that we do not make use of the number of free variables during the proof. Alas, we will have to pay this price when defining the types for System F later.

[9] o[B,C|Type][f:B->C][A|Type][g:A->B] = [a:A]f (g a) is composition.

We define the operations weakening `weak` [10] (= introduction of dummy variables) and substitution `subst`: [11]

```
weak_var : Nat -> Nat -> Nat

rec weak_var zero     i = (succ i)
  | weak_var (succ l) zero = zero
  | weak_var (succ l) (succ i) = succ (weak_var l i)

weak : Nat->Tm->Tm

rec weak l (var i) = var (weak_var l i)
  | weak l (app M N) = app (weak l M) (weak l N)
  | weak l (lam M) = lam (weak (succ l) M)

[weak0 = weak zero : Tm -> Tm];

subst_var : Nat->Nat->Tm->Tm

rec subst_var zero zero N = N
  | subst_var zero (succ i) N = var i
  | subst_var (succ l) zero N = var zero
  | subst_var (succ l) (succ i) N = weak0 (subst_var l i N)

subst : Nat->Tm->Tm->Tm

rec subst l (var i) N = subst_var l i N
  | subst l (app M M') N = app (subst l M) (subst l N)
  | subst l (lam M) N = lam (subst (succ l) M N)

[subst0 = subst zero];
```

The first parameter indicates the number of bound variables - `weak0` and `subst0` are defined as abbreviations. i.e. `subst0 M N` substitutes the free variable with index 0 in `M` by `N`. [12] Consider the following example:

```
subst0 (lam (app (var zero) (var (succ zero))))
       (app (lam (var zero)) (var zero))
```

evaluates to

```
lam (app (var zero) (app (lam (var zero)) (var (succ zero))))
```

[10] Huet [Hue92] calls the operation *lift*. I prefer the name *weak* because it corresponds to the notion of weakening in the typed case.

[11] The operations `weak_var` and `subst_var` are just auxiliary.

[12] Although we only use `weak0` and `subst0` in the following definition we really have to *export* the general versions because we have to use them whenever we want to prove anything about substitution or weakening in general (i.e. for terms containing λ-abstractions).

This corresponds to $(\lambda x.xy)[y := (\lambda z.z)x] \equiv_\alpha \lambda x'.x'(\lambda z.z)x$.

The section called *substitution and weakening laws* is the place where the formal proof becomes much more complicated than the paper version. When doing the proof on paper we are using a number of intuitive ideas about weakening and substitution which have to be made explicit here. It was this place where I unexpectedly spent quite a lot of time when doing the proof. However, to *understand* the proof it is not necessary to understand all the subtleties of de Bruijn's notation - we can apply our intuition about bound variables and substitution.

We define the one-step reduction relation [13] by the following inductive type:

```
mu[Step:Tm->Tm->Set](
        beta : {M,N:Tm}Step (app (lam M) N) (subst0 M N),
        app_l : {M,M',N:Tm}(Step M M')->(Step (app M N) (app M' N)),
        app_r : {M,M',N:Tm}(Step M M')->(Step (app N M) (app N M')),
        xi : {M,N:Tm}(Step M M')->(Step (lam M) (lam M')) )
```

This amounts to translating the usual Horn clauses defining the reduction relation into the constructors for an inductive type.

3.5 snorm.l

This is actually the core of the proof. One of the main technical contributions which simplify the formalization of the proof is the definition of the predicate strongly normalizing by the following inductive type:

```
mu[SN:Tm->Set](
    SNi : {M:Tm}({N:Tm}(Step M N)->(SN N))->(SN M))
```

In other words: we define SN as the set of elements for which Step is well founded. This seems to be related to the use of well founded relations for general recursion [Pau86].

More intuitively: SN holds for all normal forms because for them the premise of SNi is vacuously true. Now we can also show that all terms which reduce in one step to a normal form are SN and so on for an arbitrary number of steps. On the other hand these are all the terms for which SN holds because SN is defined inductively.

We use the non-dependent version of the recursor

```
RecSNN : {P:Tm->Type}
         ({M:Tm}({N:Tm}(Step M N)->SN N)->({N:Tm}(Step M N)->P N)->P M)
         ->{M|Tm}(SN M)->P M];
```

to simulate induction over the length of the longest reduction of a strongly normalizing term — in terms of [GLT89] this is induction over $\nu(M)$. Observe that we never have to

[13] We are going to define Red (the reflexive, transitive closure of Step) later (page 15). Note, however, that we never need it for the strong normalization proofs.

formalize the concept of the length of a reduction or to define the partial function ν. It is also interesting that the important property that SN is closed under reduction shows up as the destructor for this type (SNd).

In the definition of *Candidates of Reducibility* `CR:(Tm->Set)->Set` we follow [GLT89]:

```
[neutr[M:Tm] = {M':Tm}not (EQ (lam M') M)];

[P:Tm -> Set]

[CR1 = {M|Tm}(P M)->(SN M)]
[CR2 = {M|Tm}(P M)->{N:Tm}(Step M N)->(P N)]
[CR3 = {M|Tm}(neutr M)->
         ({N:Tm}(Step M N)->(P N))->(P M)]
[CR = CR1 /\ CR2 /\ CR3];

Discharge P;
```

We define `neutr` as the set of terms which are not generated by the constructor for the arrow type — λ. [14] `CR1` places an upper bound on candidates: they may only contain strongly normalizing terms. `CR2` says that candidates have to be closed under reduction and `CR3` is essentially `SNi` restricted to neutral terms.

The essence of this definition lies in the possibility of proving the following lemmas:

`CR_nonEmpty` *Candidates contain all variables*

```
{P:Tm->Set}(CR P)->{i:Nat}P (var i);
```

We need this not only for the following lemmas, but also for the final corollary when we want to deduce strong normalization from soundness for non empty contexts.

This is a trivial consequence of `CR3` because variables are neutral terms in normal form.

`CR_SN` *There is a candidate set*

```
CR SN
```

The choice is arbitrary but the simplest seems to be `SN`. The proof is trivial: just apply `SNd` for `CR2` and `SNi` for `CR3`.

`CR_ARR` *Candidates are closed under the semantic interpretation of arrow types.*

```
{P,R:Tm->Set}(CR P)->(CR R)->(CR (ARR P R))
```

where

[14] If we generalize this definitions to systems with inductive types we have to include their *constructors* as well.

```
[ARR[P,R:Tm->Set] = [M:Tm]{N:Tm}(P N)->(R (app M N))];
```

This proof of CR3 is actually quite hard and requires a double induction using RecSNN which corresponds to the reasoning using $\nu(N)$ in [GLT89].

Lam_Sound *The rule of arrow introduction (*Lam*) is semantically sound for candidate sets.*

```
{P,R:Tm->Set}(CR P)->(CR R)->
    {M:Tm}({N:Tm}(P N)->(R (subst0 M N)))
    ->(ARR P R (lam M));
```

Observe that we could not have proven this lemma for arbitrary subsets of SN. The proof requires a nested induction using RecSNN which corresponds to an induction over $\nu(M) + \nu(N)$.

CR_PI *Candidates are closed under the interpretation of* Π *types*

```
{F:(Tm->Set)->(Tm->Set)}
       ({P:Tm->Set}(CR P)->(CR (F P)))
       -> (CR (PI F));
```

where

```
[PI[F:(Tm->Set)->(Tm->Set)] =
       [M:Tm]{P:Tm->Set}(CR P)->(F P M)];
```

This lemma is only required for strong normalization for System F. At this point we really need impredicativity for the proof. However, it is interesting to observe how simple this lemma is technically: We do not apply any induction — we just have to show that CR is closed under arbitrary non-empty intersections.

3.6 simple.l

We define simple types (Ty) with one base type:

```
mu[Ty:Set](ty0 : Ty, arr : Ty -> Ty -> Ty)
```

Contexts are just vectors of types and derivations can be defined as an inductive type again:

```
mu[Der:{n|Nat}(Con n)->Tm->Ty->Set](
        Var : {n|Nat}{G:Con n}{i:Fin n}
                Der G (var (Fin2Nat i)) (v_nth G i)
        App : {n|Nat}{G|Con n}{s,t|Ty}{M,N|Tm}
                (Der G M (arr s t))
                -> (Der G N s)
                -> (Der G (app M N) t)
        Lam :  {n|Nat}{G|Con n}{s,t|Ty}{M|Tm}
                (Der (v_cons s G) M t)
                -> (Der G (lam M) (arr s t)) )
```

In the rule `Var` we use `Fin` because this rule is only applicable to integers smaller than the length of the context. At this place it would have been nicer if we had used the alternative representation of terms as a dependent type because then we could have used `i` directly. Here we have to coerce it to a natural number first (`Fin2Nat`) because `var` requires `Nat` as an argument.

3.7 sn-simple.l

We first define an interpretation of simple types as sets of terms:

```
Int : Ty->(Tm->Set)

rec Int ty0 = SN
  | Int (arr s t) = ARR (Int s) (Int t)
```

The interpretation of the type constant is an arbitrary candidate because there is no closed term of this type anyway (i.e. there is no introduction rule). Now using `CR_SN` and `CR_ARR` it follows that all interpretations of types are candidates (`CR_Int`).

It is pretty straightforward to extend this to an interpretation of judgements i.e. pairs of types and contexts (`Mod`). The idea is that `Mod G M T` holds iff we substitute all the variables in `M` by terms of the corresponding interpretation of the type in `G` we end up with an element of `T`:

```
Mod : {n|Nat}(Con n)->Tm->Ty->Set

rec Mod zero empty M T = (Int T) M
  | Mod (succ n) (v_cons S G) M T =
       {N:Tm}(Int S N)->(Mod G (subst0 M (rep_weak0 N n) T)
```

`rep_weak0` is the repeated application of `weak0`. It is necessary to apply repeated weakening here because we do not get parallel substitution by iterating `subst0` — i.e. in

`subst0 (subst0 M N1) N2` the first free variable in `N1` is also substituted by `N2` which is not intended. Correct is `subst0 (subst0 M (weak0 N1)) N2`.

We now have to show soundness (`Int_sound`), i.e. that `Der G M T` implies `Mod G M`. In the formal version this requires quite an intricate induction. However, the idea is just to apply `Lam_sound`. Indeed, the soundness of arrow elimination `App` is trivial just from the way `ARR` is defined.

Finally we have to combine `Int_Sound` with `CR_Int` to conclude (`snorm`):

```
{n|Nat}{G|Con n}{M|Tm}{T|Ty}(Der G M T)->(SN M)
```

3.8 f.l

To define the syntax of System F turns out to be quite a bit more difficult than the corresponding task for simply typed λ calculus. Sure, we could just use the type of untyped λ terms as a representation of the types of System F. But in contrast to the untyped λ terms the number of free variables of a System F type is quite essential — it will determine the semantic interpretation of types later. Therefore I decided to use the following dependent type as a representation for types of System F [15] — it corresponds to the alternative definition of untyped λ terms discussed earlier:

```
mu[Ty:Nat->Set](t_var : {n|Nat}(Fin n) -> (Ty n),
                arr   : {n|Nat}(Ty n) -> (Ty n) -> (Ty n),
                pi    : {n|Nat}(Ty (succ n)) -> (Ty n) )
```

When defining weakening and substitution for `Ty` we observe that the types actually tell us how this operations behave on free variables:

```
t_weak : {l|Nat}(Tm (add l n))->(Ty (succ (add l n)))
t_subst : {l:Nat}(Ty (add (succ l) n))->(Ty n)->(Ty (add l n))
```

Although these functions are operationally equivalent to `weak` and `subst` we have to put in more effort to implement them. We do this by deriving some special recursors (`RecTy1`,`RecTy2`) from the standard recursor (`RecTy`). It seems that we could save a lot of effort here by using Thierry Coquand's idea of considering definitions by pattern matching as primitive [Coq92b].

Analogue to `simple` we define derivations:

```
mu[Der:{m,n|Nat}(Con m n)->Tm->(Ty m)->Set](
        Var : {m,n|Nat}{G:Con m n}{i:Fin n}
                Der G (var (Fin2Nat i)) (v_nth i G)
        App : {m,n|Nat}{G|Con m n}{s,t|Ty m}{M,N|Tm}
                (Der G M (arr s t))
```

[15] Note that the names used for simply typed λ calculus and System F overlap. This just implies that we can not load both proofs at the same time.

```
        -> (Der G N s)
        -> (Der G (app M N) t)
Lam : {m,n|Nat}{G|Con m n}{s,t|Ty m}{M|Tm}
        (Der (v_cons s G) M t)
        -> (Der G (lam M) (arr s t))
Pi_e: {m,n|Nat}{G|Con m n}{s:Ty (succ m)}{t:Ty m}{M|Tm}
        (Der G M (pi s))
        -> (Der G M (t_subst0 s t))
Pi_i: {m,n|Nat}{G|Con m n}{s:Ty (succ m)}{M|Tm}
        (Der (v_map t_weak0 G) M s)
        -> (Der G M (pi s)) )
```

`v_map t_weak0 G` means that all the types in `G` are weakened — this is equivalent to the usual nasty side condition in the standard definition of Π-introduction. It is nice to observe how well the types of `t_subst0` and `t_weak0` fit for the definition of the rules.

3.9 sn-f.l

We proceed along the same lines as for `sn-simple.l` but everything gets more complicated due to the fact that types contain variables. We define the interpretation function using large vectors:

```
Int : {m|Nat}(Ty m)->(VEC (Tm->Set) m)->(Tm->Set)

rec Int|m (t_var i) = [v:VEC (Tm->Set) m]V_nth i v
  | Int|m (arr s t) = [v:VEC (Tm->Set) m]ARR (Int s v) (Int t v)
  | Int|m (pi t)    = [v:VEC (Tm->Set) m]PI [P:Tm->Set]Int t (V_cons P v)
```

That is we interpret types with free variables as functions from sets of terms to a set of terms. `CR_Int` now expresses that every interpretation preserves candidates:

```
{m|Nat}{t:Ty m}{v:VEC (Tm->Set) m}
        ({i:Fin m}CR (V_nth i v))
        -> (CR (Int t v));
```

To show soundness for System F we have to show that substitution and weakening for types are interpreted correctly (`t_weak_lem` and `t_subst_lem`). These semantically trivial lemmas (nobody would write their proofs down on paper) turn out to be some of the technically most complicated proofs in the whole formalization.

When we want to derive strong normalization from soundness (as for the simply typed λ calculus) we have to interpret the free type variables. We just use `SN` because we have already shown that this is a candidate. It is interesting to observe that both simply typed λ calculus and System F require the fact that `CR` is not empty but in both cases the choice seems to be arbitrary.

4 Discussion of the formalization

I will add a few more comments about alternatives or extensions of the proof.

4.1 Extracting a normalization function

We now know that every typable term is strongly normalizing but how can we compute its normal form? Indeed, it is possible to use RecSNN to do this. This seems to be a case where it is actually more straightforward to give a proof that every strongly normalizable term has a normal form than to program it directly.

To specify normalization we need a notion of reduction (Red) — which is just the transitive reflexive closure of step:

```
mu[Red:Tm->Tm->Set](
        r_refl : {M:Tm}(Red M M),
        step : {M1,M2,M3|Tm}(Step M1 M2)->(Red M2 M3)->(Red M1 M3) )
```

Now we want to show norm_lem:

```
{M:Tm}(SN M)->Ex[M':Tm](Red M M')/\(nf M')
```

which can be done using RecSN — it turns out that we need decidability of normal form (nf) as a lemma:

```
{M:Tm}(nf M)\/(Ex[M':Tm]Step M M')
```

Actually this is even stronger because it also gives us a choice of a reduct for terms not in normal form (this is the point where we specify the strategy of reduction).

If we use the strong sum to implement Ex, instead of the weak impredicative encoding, we can use norm_lem to derive:

```
norm    : {M:Tm}(SN M)->Tm
norm_ok : {M:Tm}{p:SN M}(Red M (norm M p)) /\ (nf (norm M p))
```

4.2 Saturated Sets

In many strong normalization proofs the notion of *Saturated Sets* is used instead of *Candidates of Reducibility* (e.g. [Luo90], [B+91]). How much would we have to change to adopt this method ? And is it easier to use Saturated Sets? Although I have not done the formalization, it is easy to answer the first question: Most of the proof would remain unchanged because Saturated Sets just correspond to an alternative induction hypothesis, that is we would have to replace CR by SAT and proving the same lemmas (i.e. this would involve changing the second half of snorm.1).

To address the second question let us have a look at SAT — following [Luo90] p. 93 we first define weak head reduction (W_Hd_Step) [16] and the predicate *base term* (BaseTerm):

[16] Luo uses the term *key redex* here.

```
mu[W_Hd_Step:Tm->Tm->Set](
        wh_beta : {M,N:Tm}W_Hd_Step (app (lam M) N) (subst0 M N),
        wh_app_l : {M,M',N:Tm}(W_Hd_Step M M')
                        ->(W_Hd_Step (app M N) (app M' N)) )

mu[BaseTerm : Tm -> Set](
        bt_var : {i:Nat}BaseTerm (var i),
        bt_app : {M,N|Tm}(BaseTerm M)->(SN N)->(BaseTerm (app M N)) )
```

We can now translate Luo's definition into:

```
[P:Tm -> Set]

[SAT1 = {M|Tm}(P M)->(SN M)]
[SAT2 = {M|Tm}(BaseTerm M)->(P M)]
[SAT3 = {M|Tm}(SN M)->(Ex[N:Tm](W_Hd_Step M N)/\(P N))->(P M)];

Discharge P;
```

Actually I believe [17] that we can dispense with BaseTerm and use the following (equivalent) definition:

```
[P:Tm -> Set]

[SAT'1 = {M|Tm}(P M)->(SN M)]
[SAT'2 = {M|Tm}(neutr M)->(SN M)->({N:Tm}(W_Hd_Step M N)->(P N))->(P M)]
[SAT' = SAT'1 /\ SAT'2];

Discharge P;
```

Luo shows that CR P implies SAT P ([Luo90], page 95) and remarks that the converse does not hold because Saturated Sets do not have to be closed under reduction. An example is the set of all terms whose weak head normal form is neutral or equal to $\lambda x.II$ which is saturated but not closed under reduction.

If we want to show CR_ARR and Lam_sound the main problem is to show the side condition for SAT'2 (or SAT3): SN M. This involves reasoning with RecSN in a manner very similar to the original proof. The conclusion therefore is that it does not seem to be easier to use Saturated Sets.

4.3 Reduction for Church terms

We have only done the proofs for the Curry Style systems — so one obvious question is how hard it would be to extend this proof to the Church Style presentation, i.e. to terms with explicit type information. Usually Church terms only contain explicit

[17] This means I have only proven it on paper.

type information in the abstractions; here we will go a little bit further and consider derivations without subject. [18]

To make this a bit more precise — let us define typed terms (`Tm'`) (or Church terms) [19] for simply typed λ calculus as follows:

```
mu[Tm':{n|Nat}(Con n)->Ty->Set](
        var' : {n|Nat}{G:Con n}{i:Fin n}
                 Tm' G (v_nth i G)
        app' : {n|Nat}{G|Con n}{s,t|Ty}
                 (Tm' G (arr s t))
                 -> (Tm' G s)
                 -> (Tm' G t)
        lam' :  {n|Nat}{G|Con n}{s:Ty}{t|Ty}
                 (Tm' (l_cons s G) t)
                 -> (Tm' G (arr s t))
```

We can easily recover the subject of the derivation by striping the types:

```
subject : {n|Nat}{G|Con n}{T|Ty}(Tm' G T)->Tm
```

We expect `subject` to have the following properties:

```
{n|Nat}{G|Con n}{T|Ty}{M:Tm' G T}Der G (subject M) T
{n|Nat}{G|Con n}{T|Ty}{M:Tm}(Der G M T)->Ex[M':Tm' G T]EQ M (subject M')
```

Assuming substitution for `Tm'` [20]

```
subst0' : {m|Nat}{G|Con m}{S,T|Ty}
            (Tm' (v_cons S G) T)->(Tm' G S)->(Tm' G T)
```

we define reduction for typed terms:

```
mu[Step':{m|Nat}{G|Con m}{T|Ty}(Tm' G T)->(Tm' G T)->Set](
        beta' : {m|Nat}{G|Con m}{S,T|Ty}{M:Tm' (v_cons S G) T}{N: Tm' G S}
                  Step' (app' (lam' G S M) N) (subst0' M N),
        app_l' : {m|Nat}{G|Con m}{S,T|Ty}{M,M':Tm' G (arr S T)}{N:Tm' G S}
                  (Step' M M')
                  ->(Step' (app' M N) (app' M' N)),
        app_r' : {m|Nat}{G|Con m}{S,T|Ty}{M,M':Tm' G S}{N:Tm' G (arr S T)}
                  (Step' M M')
```

[18] The usual presentation of Church terms is important in the context of *type reconstruction*. However, my impression is that Church presentations got too much attention as Calculi on their own right.

[19] Although we can supress some of the explicit type information the notation is still very clumsy — i.e. $\lambda x : 0.x$ becomes: `lam' empty ty0 (var' (v_cons ty0 empty) (f_zero zero))`

[20] This is equivalent to proving the substitution lemma which is the core of the *subject reduction* property. However, the proof has just the same structure as the definition of `t_subst`.

```
                ->(Step' (app' N M) (app' N M')),
        xi' : {m|Nat}{G|Con m}{S:Ty}{T|Ty}{M,M':Tm' (v_cons S G) T}
                (Step' M M')
                ->(Step' (lam' G S M) (lam' G S M')) )
```

and we should be able to show that `subject` commutes with reduction, i.e.:

```
{n|Nat}{G|Con n}{T|Ty}{M,N:Tm' G T}
              iff (Step' M N) (Step (subject M) (subject N));
```

The idea behind this is that the subject of a derivation tree, i.e. its skeleton, precisely captures its behaviour wrt reduction.

It is worthwhile to note that the reasoning above does not directly generalize to System F. The reason is that we have second order redexes in System F's derivation trees which are not reflected by redexes in untyped terms. The usual way to get around this is to argue that it is pretty easy to see that the second order reductions are always terminating. Another — maybe cleaner way — would be to actually extend the notion of untyped term s.t. for second order reduction is taken care of. This would mean to extend the type `Tm` [21] by:

```
mu[Tm:Set](...,
           T_Lam : Tm -> Tm,
           T_App : Tm -> Tm)
```

where `T_Lam` *does not* act as a variable binder. We may now extend `Step` by the following trivial case:

```
mu[Step:Tm->Tm->Set](...,
        Beta : {M:Tm}(Step (T_App (T_Lam M)) M) )
```

For this extended notion of untyped reduction it should be possible to show that `subject` preserves reduction. It seems that to show strong normalization for this calculus we actually have to prove a lemma to show that `Pi_i` is semantically sound (which was completely trivial so far). In this regard doing the proof for the Curry presentation is actually a little bit of cheating.

5 Future work

My original motivation to do this formalization was actually to understand better how to do extend the standard proofs to systems with inductive types, i.e. I was interested in checking whether a proof on paper [Alt92] was correct. [22]

[21] It may just be a curiosity but this version of untyped terms correspond to (a special case of) partial terms. In [Pfe92] it is shown that type inference for partial terms is undecidable which is still open for the usual notion of untyped terms.

[22] I still have not done this because I got a little bit sidetracked as one sees.

In [GLT89] strong normalization for System T is proven as an extension of simply typed λ calculus. Here the type **Nat** is interpreted by **SN**. This seems to be a rather arbitrary choice and the price one has to pay for it is that one has to do an additional induction to show that the elimination rule for **Nat** is sound. [23] This technique does not seem to generalize to non-algebraic types (at least one would need induction over an ordinal greater than ω) and in [Alt92] a technique is introduced which avoids this problem essentially by using an impredicative trick on the level of semantics and applied to the System T enriched by a type of notations for countable ordinals (We call this system T^{Ω}). Here the usual structure of the proof is preserved and the soundness of elimination rules is trivial. I hope to finish a formal version of a proof for arbitrary inductive types based on this idea in due course. [24]

6 Why shall we formalize proofs ?

Strong normalization for System F has been proven the first time in Girard's thesis [Gir72] and since then a number of versions of the proof have been published (e.g. [GLT89],[B+91],[Gal90]). Therefore we may say that this proof is well understood and nobody should doubt its correctness. Insofar the formalization I have done may rather serve as a demonstration that it is possible to formalize a fairly complex proof than as a confirmation of the well accepted fact that System F is strongly normalizing.

However, doing this proof in LEGO was quite a rewarding experience. First of all I finished the proof in much shorter time than expected. It took me two weeks from editing the first definition until proving the last lemma. Surely I do not count the time to understand the proof and to learn how to use LEGO. When I was doing the proof I found out at some points that I had not really understood it: at one point I had to apply a lemma a second time at another I found that my original reasoning was actually wrong, i.e. I could not prove a lemma because it was simply false. In the end everything worked out and completing the proof was like doing a big puzzle. I learnt a lot about strong normalization proofs and I believe that I understand now better what parts of the proof depend on what properties of the λ calculus and also where one should change something for different calculi.

Given that formalizing the argument after having understood it was not so much of an additional effort, I believe that *Computer Aided Formal Reasoning* (CAFR) may serve as a useful tool in mathematical research in future. It does not only raise your's and other people's confidence in the correctness of the argument it also helps you to understand it better and to understand its fine structure. I also observed that sometimes it is easier to completey formalize a very technical lemma than to convince yourself beyond reasonable doubt that it is true by completely understanding it. In this case the formalization may even save time. Formalizing an argument also forces you to use very clean definitions and to structure the proof neatly.

Yet, there is still another point in favour of CAFR. Assume somebody has done a fairly intricate proof of Fermat's last theorem. You do not want to spend the time to completely

[23] See [GLT89], p.49, case 4. of the proof.

[24] Nax Mendler [Men87][Men88] shows strong normalisation for calculi with inductive types in a different formulation using a similar technique.

understand the proof but you would like to know wether it is right. If the use of proof checker and formal proofs becomes widespread you may just ask for a formal version of the proof. Then you feed it into a proof checker you trust and all you have to check is wether you believe that the statement of the theorem is adequate. In this way computer may actually rationalize mathematical research.

Acknowledgements

Thanks to Randy Pollack who got me to do this by not believing me that it could be done in the time I claimed (he was right). I would also like to thank the following people for helpful discussions, help in proof reading, etc: Matt Fairtlough, Martin Hofmann, Zhaohui Luo, Benjamin Pierce and Benjamin Werner.

References

[Alt90] Thorsten Altenkirch. Impredicative representations of categorical datatypes, thesis proposal, October 1990.

[Alt92] Thorsten Altenkirch. Strong normalisation for T^{Ω}. unpublished note, July 1992.

[B+91] Henk Barendregt et al. Summer school on λ calculus, 1991.

[Ber91] Stefano Berardi. Girard's normalisation proof in LEGO. unpublished draft, 1991.

[Coq92a] Catarina Coquand. A strong normalisation proof for simply typed λ calculus in ALF. In *Workshop on Logical Frameworks*, 1992. (to appear).

[Coq92b] Thierry Coquand. Patterns with dependent types. In *Workshop on Logical Frameworks*, 1992.

[CP89] Thierry Coquand and Christine Paulin. Inductively defined types. In Peter Dybjer et al., editors, *Proceedings of the Workshop on Programming Logic*, 1989. Preliminary version.

[D+91] Gilles Dowek et al. *The Coq Proof Assistant User's Guide.* INRIA-Rocquencourt — CNRS-ENS Lyon, 1991. Version 5.6.

[dB72] 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.*, 34, 1972.

[Gal90] J.H. Gallier. On Girard's "Candidats de Reducibilité". In Piergiogio Oddifreddi, editor, *Logic and Computer Science.* Academic Press, 1990.

[Gir72] J.-Y. Girard. Interprétation fonctionelle et élimination des coupures de l'arithméthique d'ordre supérieur. Thése de Doctorat d'Etat, Université de Paris VII, 1972.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[Hof92] Martin Hofmann. Formal development of functional programs in type theory — a case study. LFCS Report, University of Edinburgh, 1992. (to appear).

[Hue92] Gérard Huet. Initation au lambda calcul. Lecture notes, 1992.

[LP92] Zhaohui Luo and Robert Pollack. Lego proof development system: A user's manual. LFCS report ECS-LFCS-92-211, University of Edinburgh, 1992.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[Mag92] Lena Magnusson. The new implementation of ALF. In *Workshop on Logical Frameworks*, 1992. (to appear).

[Men87] N.P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of LICS 87*, 1987.

[Men88] N.P. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, 1988.

[NPS90] Begt Nordström, Kent Petersson, and Jan Schmith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.

[Pau86] L.C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 1986.

[Pfe92] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. Technical Report CMU-CS-92-105, Carnegie Mellon University, January 1992.

[Smi87] Jan M. Smith. The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. Technical Report 31, Programming Methodology Group, University of Goteborg and Chalmers University of Technology, 1987.

[Wer92] Benjamin Werner. A normalisation proof for an impredicative type system with large eliminations over integers. In *Workshop on Logical Frameworks*, 1992. (to appear).

A How to get the LEGO code?

The Lego files can be obtained on e-mail request (alti@dcs.ed.ac.uk) or by anonymous internet ftp from host `ftp.dcs.ed.ac.uk` (address 129.215.160.5), directory `pub/alti`, file `snorm.tar.Z` (this is a compressed tar file which contains a directory including a file `README` for further information). Here is a sample dialog:

```
ftp ftp.dcs.ed.ac.uk
Name: anonymous
Password: your email-address
ftp> binary
ftp> cd pub/alti
ftp> get snorm.tar.Z
ftp> bye
```

After the file is transferred it should be uncompressed and then extracted using `tar`. For example: `zcat snorm.tar.Z | tar fox -`.