A Formalization of the Strong Normalization Proof for System F in LEGO [*][†]

Thorsten Altenkirch

December 8, 1992

Abstract

We describe a complete formalization of a strong normalization proof for the Curry style presentation of System F in LEGO. The underlying type theory is the Calculus of Constructions enriched by inductive types. The proof follows Girard et al [GLT89], i.e. we use the notion of *candidates of reducibility*, but we make essential use of general inductive types to simplify the presentation. We discuss extensions and variations of the proof: the extraction of a normalization function, the use of *saturated sets* instead of candidates, and the extension to a Church Style presentation. We conclude with some general observations about Computer Aided Formal Reasoning.

1 Introduction

I am going to describe a complete formalization of a strong normalization proof for System F in LEGO [LP92]. The proof[1] uses the tactics provided by the LEGO system. However, in the end we can extract a typed λ-term which represents the proof. The proof is complete, i.e. there are no non-logical axioms used.[2] This implies in particular that the proof is intuitionistic. As a consequence of this we can extract a normalization function together with its verification from the proof.

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

[*]When doing this research I have been supported by a SIEMENS studentship. This research was also partially supported by the ESPRIT BRA on Logical Frameworks and a SERC grant.

[†]This is a revision of the LFCS report ECS-LFCS-92-230 “Brewing Strong Normalization Proofs with LEGO”.

[1]The complete proof text can be obtained by anonymous ftp from `ftp.dcs.ed.uk` or by email request. The directory is `pub/alti`; the file is `snorm.tar.Z`.

[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].

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.

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 over the Pure Calculus of Constructions. We also diverge from Luo's proposal to use the predicative universes (`Type`) for computations, instead using 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.

Many formal proofs in the Calculus of Constructions use the so called *impredicative encodings* of inductive types (e.g. see [Alt90]). These encodings have a number of disadvantages, in particular 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

The basic logical connectives (/\,\/,`not` and `Ex`) are defined using an impredicative encoding. However, instead of Leibniz Equality we define propositional equality `EQ` as an inductive type in the same way as in Martin-Löf Type Theory.

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.

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 inferred when the function is applied.
$\Pi_{x:A} B$	`{x:A}B` *or* `{x|A}M`	`{x|A}B` is used for the typing of `[x|A]M`.
MN	`M N` *or* `M|N`	`M|N` is used to supply an implicit argument.

Figure 1: Syntax of terms in LEGO

2.3 Inductive types

So far inductive types have not been implemented in the LEGO system but we can use 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 into the proof. This is then expanded into LEGO code — I will explain this by a simple example:

```
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 constructor as constants. We also add an elimination constant:

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

We declare typed rewriting rules which correspond to primitive recursion on natural numbers. 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 (mechanically) translated into the following LEGO code using the recursor (here we use a derived non-dependent recursor `RecNatN`[3] to simplify the typing):

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

[3]We adopt the convention that *R*`N` stands for the non-dependent version of recursor *R*.

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

We can only define functions by primitive recursion in this way, but note that we get more than the usual primitive recursive functions 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))
```

or the family of finite sets:

```
mu[Fin:Nat->Set](f_zero:{n:Nat}Fin (succ n),
                 f_succ:{n|Nat}(Fin n)->(Fin (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`[4] is the type of sequences of type `A` of length `3`. Finite sets are a representation of subsets of natural numbers less than a certain number, i.e. `Fin` n corresponds to $\{i \mid i < n\}$.

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. 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* lookup function for vectors:[5]

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

rec v_nth|(succ n) (f_zero n) (v_cons a _) = a
  | v_nth|(succ n) (f_succ i) (v_cons _ l) = v_nth|n i l
```

Using these error-free functions not only simplifies the verification of functions using vectors; it also allows, in principle a more efficient compilation of code involving dependent types.[6]

Another use of dependent inductive types is the definition of predicates as the initial semantics of a set of Horn clauses. 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)))
```

[4]The official LEGO syntax for this is `Vec A (succ (succ (succ zero)))`.

[5]I.e. `v_nth (f_succ (f_zero 3)) : (Vec A 5)->A` extracts the second element out of a sequence of five.

[6]The idea to use dependent types to avoid run-time-errors was first proposed by Healfdene Goguen to me.

3 A guided tour through the formal proof

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

3.1 The untyped λ-calculus

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)
```

We define the operations weakening `weak : Nat->Tm->Tm`[7] (introduction of dummy variables) and substitution `subst : Nat->Tm->Tm->Tm` by primitive recursion over the structure of terms.

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`.[8]

In the course of the proof we need a number of facts about weakening and substitution. An example is the following proposition stating that under certain conditions we can interchange substitution and weakening:

```
{l',l:Nat}{M,N:Tm}(LE l' l)
->(EQ (subst (succ l) (weak l' M) N) (weak l' (subst l M N)))
```

Stating and proving this sort of lemma takes up a lot of time when doing the formalization whereas in the informal proof one would just appeal to some intuition about substitution and bound variables.

We define the one-step reduction relation[9] 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.

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

[8]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).

[9]We are going to define `Red` (the reflexive, transitive closure of `Step`) later (section 4.1). Note, however, that we never need it for the strong normalization proof.

3.2 Strong Normalization

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:[10]

```
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.

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 will use the non-dependent version of the recursor [11]

```
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 formalize the concept of the length of a reduction or to define the partial function ν. [12] It is also interesting that the important property that `SN` is closed under reduction shows up as the destructor for this type (`SNd`).

3.3 System F

The type expressions of System F have essentially the same structure as untyped λ-terms. However, in contrast to the definition of `Tm` we will use a dependent type here, which makes the number of free variables explicit. This turns out to be useful when we define the semantic interpretation of types later.[13]

```
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) )
```

`Ty` i represents type expressions with i free variables.

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

[10]It is interesting to note that this inductive type is not algebraic or equivalently does not correspond to a specification by a set of Horn formulas. However, the requirement of (strict) positivity as stated in [CP89] is fulfilled.

[11]This corresponds to the principle of Noetherian Induction [Hue80].

[12]Note that *bounded* and *noetherian* coincide for β-reduction because it is finitely branching (König's lemma).

[13]We could have used a dependent type for `Tm` as well, but we never need to reason about the number of free variables of an untyped term.

```
t_weak : {l:Nat}(Ty (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 from the standard recursor.[14]

We now define contexts and derivations as:

```
[Con[m:Nat] = Vec (Ty m)];

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))
                -> (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)) )
```

In the rule `Var` we use `Fin` because this rule is only applicable to integers smaller than the length of the context. Here we have to coerce it to a natural number first (`Fin2Nat`) because `var` requires `Nat` as an argument.

`v_map t_weak0 G` means that all the types in `G` are weakened — this is equivalent to the usual 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.

3.4 Candidates

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*.

[14]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].

We have to find a family of sets of terms,[15] here called the *Candidates of Reducibility*, such that we can show the following things:

1. Every Candidate only contains strongly normalizing terms.
2. For every operation on types we can define a semantic operation on sets of terms such which is closed under candidates. Another way to express this is to say that the Candidates constitute a logical predicate.
3. Candidates are sound, i.e. every term which has a type is also in the semantic interpretation of the type.

Putting these things together we will obtain that every typable term is strongly normalizing.

In the definition of *Candidates of Reducibility* `CR:(Tm->Set)->Set` we follow [GLT89]:[16]

```
[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 — `lam`.[17] `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_var` *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.

[15]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`.

[16]`[P:Tm->Set] ... Discharge P;` means that `P` is λ-abstracted from all definitions in between.

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

`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

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

The proof of `CR3` for `ARR P R` is actually quite hard and requires an induction using `RecSNN` which corresponds to the reasoning using $\nu(N)$ in [GLT89].

`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)];
```

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.

`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 proved this lemma for arbitrary subsets of `SN`. The proof requires a nested induction using `RecSNN` which corresponds to an induction over $\nu(M) + \nu(N)$.

3.5 Proving strong normalization

We now have all the ingredients for the proof, we just have to put them together.

We proceed by defining an interpretation function. Types are interpreted by functions from sequences of sets of terms to sets of terms, the length of the sequence depending on the number of free type variables:[18]

```
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))
```

We can show by a simple induction that every interpretation of a type preserves candidates (`CR_Int`) by exploiting `CR_ARR` and `CR_PI` .

We extend this to an interpretation of judgements, i.e. pairs of types and contexts (`Mod`). The idea is that `Mod G M T v` holds iff by substituting all the variables in `M` by terms of the corresponding interpretation of the types in `G` we end up with an element of `Int T v`:[19]

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

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

We use `Mod` to state soundness (`Int_Sound`), i.e. that `Der G M T` implies `Mod G M T v` if all free type variables are interpreted by candidates:

```
{m,n|Nat}{G|Con m n}{M|Tm}{T|Ty m}(Der G M T)->
       {v:VEC (Tm->Set) m}({i:Fin m}CR (V_nth i v))
       -> (Mod G M T v);
```

The proof of soundness proceeds by induction over derivations. Essentially we only have to apply `Lam_sound` to show that the rule `Lam` is sound. The soundness of application `App` follows directly from the definition of `ARR`. To verify soundness for the rules which are particular to System F we do not need additional properties of `CR` but we have to verify that `t_weak` and `t_subst` are interpreted correctly with respect to `Int`. Again these intuitively simple lemmas are quite hard to show formally.

To conclude strong normalization:

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

[18] We have to use another type of vectors (large vectors) `VEC:Nat->Type(0)->Type(0)` instead of `Vec:Nat->Set->Set`. Unfortunately, this sort of polymorphism cannot be expressed in the current implementation of LEGO — i.e. we have to duplicate the definitions.

[19] `rep_weak0` is the iterated application of `weak0`. It is necessary to apply weakening here because we do not get parallel substitution by a repeated application of `subst0`.

we have to put `Int_sound` and `CR_Int` together to show that every term is in the interpretation of a candidate; and by definition candidates only contain strongly normalizing terms. There are two technical complications: to show the theorem for terms with free term variables we exploit `CR_var`; to show it for a derivation with free type variables we have to supply a candidate — here we use `CR_SN`. Note that the choice is arbitrary but that it is essential that `CR` is not empty.

4 Alternatives and extensions

Apart from the first section I have not yet formalized the following ideas, but I have checked them on paper.

4.1 Extracting a normalization function

The proof not only tells us that every typable term is strongly normalizing but it is also possible to derive a function which computes the normal form. 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) )
```

and we define the predicate *normal form*:

```
[nf[M:Tm] = {M':Tm}not (Step M M')]
```

Now we want to show `norm_lem`:

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

which can be done using `RecSNN` — it turns out that we need decidability of normal form 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]). It is relatively easy to change the proof to use saturated sets: all we have to do is to replace `CR` by `SAT` and prove that it has the same properties as `CR`.

The definition of `SAT` used in the literature seems to be quite hard to formalize in Type Theory. Therefore, we use an equivalent formulation, exploiting the concept of weak head reduction:

```
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)) )
```

Now we can define `SAT` analogously to `CR`:

```
[P:Tm->Set]
[SAT1 = {M|Tm}(P M)->(SN M)]
[SAT2 = {M|Tm}(neutr M)->(SN M)
            ->({N:Tm}(W_Hd_Step M N)->(P N))->(P M)]
[SAT = SAT1 /\ SAT2];
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 strongly normalizing terms whose weak head normal form is neutral or equal to $\lambda x.\Pi$, which is saturated but not closed under reduction.

If we want to show `CR_ARR` and `Lam_sound` formally we have to use `RecSNN` in a manner similar to the original proof. Therefore using saturated sets does not seem to simplify the proof.

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. In the case of simply typed λ-calculus this is straightforward because every reduction on a typed term corresponds to one on its untyped counterpart and vice versa. However, this reasoning does not generalize to System F because here we have additional (second order) reductions on typed terms.

This problem is usually solved by arguing that the second order reductions are terminating anyway. Another way, maybe more amenable to formalization, would be to extend the notion of untyped terms and reduction:

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

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

Note that `T_Lam` does not actually bind any term-variables but corresponds to second order abstraction for typed terms; analogously `T_App` is used as a dummy type application where the type is omitted.[20]

It does not seem hard to extend the proof to this notion of untyped terms. We have to extend the notion of neutrality, and the soundness of **`Pi_i`**, which was trivial so far, has to be proved as an additional property of **`CR`**. The result for Church terms now follows by observing that for the extended notion of untyped terms reductions coincide with the typed terms.

5 Future work

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

In [GLT89] strong normalization for System T is proved 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 one consequence is that one has to do an additional induction to show that the elimination rule for **`Nat`** is sound.[21] This technique does not seem to generalize to non-algebraic types (at least one would need induction over an ordinal greater than ω). We avoid this problem by using an impredicative trick on the level of semantics and apply this technique 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 the elimination rules is trivial. I hope that I can now use the insights gained by doing this experiment to give strong normalisation proof for a system using a general notion of inductive types like the one we used to formalize this proof. [22]

6 Conclusions

The formalization of the proof in LEGO was done in a fairly short period of time — two weeks from typing in the first definition until proving the last lemma. This does not account for the time needed to understand the proof or how to use LEGO properly.

[20] It may just be a curiosity, but this version of untyped terms corresponds 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.

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

[22] This should be compared with [Men88].

When doing the formalization, I discovered that the core part of the proof (here proving the lemmas about `CR`) is fairly straightforward and only requires a good understanding of the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening. Although some may consider this as a point against formal proofs, I believe it is actually useful to check that basic definitions really reflect our intuitions about them. Many subtle errors could have been avoided this way.

However, the fact that formalizing the proof after understanding it was not so much of an additional effort seems to justify the believe that *Computer Aided Formal Reasoning* may serve as a useful tool in mathematical research in future. Using formal proofs simplifies the validation of results: we do not have do understand a subtle proof to know that the result is true, we only have to check that the formalization of the statement is correct. The process of checking whether the proof term really validates the statement can be completely mechanized. This sort of validation is not only relevant for mathematics but may prove to have a role in the area of program verification.

Acknowledgements

Thanks to Randy Pollack for implementing LEGO and for his encouragement to pursue this project. I would also like to thank the following people for helpful discussions, help in proof reading, etc: Matt Fairtlough, Healfdene Goguen, Martin Hofmann, Zhaohui Luo, Benjamin Pierce and Benjamin Werner. Thanks to the referees for their comments.

References

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

[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 proof of normalization for simply typed lambda calculus written in ALF. In *Workshop on Logical Frameworks.* BRA Types, 1992. Preliminary Proceedings.

[Coq92b] Thierry Coquand. Pattern matching with dependent types. In *Workshop on Logical Frameworks*, 1992. Preliminary Proceedings.

[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.

[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 ECS-LFCS-92-228, University of Edinburgh, 1992.

[Hue80] Gérard Huet. Confluent reductions. *JACM*, 27(4):797–821, 1980.

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

[LP92] Zhaohui Luo and Robert Pollack. The 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. Preliminary Proceedings.

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

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

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

[Wer92] Benjamin Werner. A normalization proof for an impredicative type system with large eliminations over integers. In *Workshop on Logical Frameworks.* BRA Types, 1992. Preliminary Proceedings.