Extracting a normalization algorithm in Isabelle/HOL

Stefan Berghofer

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
http://www.in.tum.de/~berghofe/

Abstract. We present a formalization of a constructive proof of weak normalization for the simply-typed λ-calculus in the theorem prover Isabelle/HOL, and show how a program can be extracted from it. Unlike many other proofs of weak normalization based on Tait's strong computability predicates, which require a logic supporting strong eliminations and can give rise to dependent types in the extracted program, our formalization requires only relatively simple proof principles. Thus, the program obtained from this proof is typable in simply-typed higher-order logic as implemented in Isabelle/HOL, and a proof of its correctness can automatically be derived within the system.

1 Introduction

The past years have seen an increasing interest in machine-assisted formalizations of the metatheory of programming languages. As a recent example, consider the PoplMark Challenge by Pierce et al. [3], which has as a goal the formalization of the basic metatheory of System $F_{<:}$ in a theorem prover, including the definition of an evaluation relation and a proof of type safety. One important aspect of this challenge is *executability*. In particular, it should be possible to generate an executable function computing the normal form of a term from the formalized definition of the evaluation relation, which allows "real" language implementations to be *tested* against the formalization.

A particularly elegant way of obtaining executable code from a formalization is the *extraction* of programs from proofs. In this article, we focus on a proof of *weak normalization* for the simply-typed λ-calculus ($\lambda_{\rightarrow}$) using the proof assistant Isabelle/HOL [16]. Weak normalization means that each well-typed term can be reduced to a term in normal form, whereas *strong normalization* means that each reduction sequence starting from a well-typed term is guaranteed to terminate. Although weak normalization follows from strong normalization as a corollary, proofs of weak normalization are interesting in their own right. If done *constructively*, a proof of weak normalization contains a particular reduction algorithm, which can be uncovered using *program extraction*.

Admittedly, the idea of extracting normalization algorithms from proofs is not completely new. It already dates back to the work of Berger [8], who describes an experiment in extracting a program from a strong normalization proof, which was

formalized using the Minlog theorem prover [7]. Like many other normalization proofs to be found in the literature, Berger's proof is based on so-called *strong computability predicates* originally introduced by Tait [18]. A term t is called *strongly computable* if

- t is of base type and t is *strongly normalizing*, or
- t is of type $\sigma \Rightarrow \tau$, and for all *strongly computable* terms s of type σ, the term $t\ s$ of type τ is *strongly computable*

Formalizing this notion of strong computability in a theorem prover poses a number of problems. Due to the *negative* occurrence of "*strongly computable*" in the second clause, the above characterization of strong computability is not admissible as an *inductive definition*. A way out of this problem is to define strong computability by recursion on the type of the term t. Unfortunately, the definition of predicates by recursion on datatypes, which is sometimes called *strong elimination*, exceeds the expressive power of theorem provers such as Minlog. For this reason, Berger does not completely formalize his proof inside the theorem prover, but performs the induction on types "on the meta level". Strictly speaking, this proof therefore does not yield a single normalization function, but a whole family of functions, which then have to be put together manually. Although Isabelle/HOL is powerful enough to define predicates such as strong computability by recursion, this is still problematic for program extraction. Since predicates in a proof become types in the extracted program, predicates defined by recursion on datatypes give rise to programs using dependent types. Such programs can neither be expressed inside Isabelle/HOL, nor can they easily be translated to functional programming languages such as ML.

As an alternative to Tait-style normalization proofs, Matthes and Joachimski [12] have given a very elegant paper proof of weak normalization for the simply-typed λ-calculus using only relatively simple proof principles. Instead of strong computability predicates, their proof uses a simple inductive characterization of β-normal terms, which turns out to be quite well suited for the purpose of program extraction.

In this article, we present a complete formalization of Matthes' and Joachimski's weak normalization proof in the theorem prover Isabelle/HOL. We show how a provably correct program can automatically be extracted from this proof using Isabelle's framework for program extraction [9,10].

Similar machine-checked formalizations have been carried out by Altenkirch [1,2], who proved strong normalization for System F using the LEGO proof assistant [13], as well as Barras and Werner [5,4] who proved decidability of type checking for the Calculus of Constructions and extracted a type checker from this proof using the Coq [6] theorem prover. A formalization of substantial parts of the metatheory of *Pure Type Systems*, also using the LEGO proof assistant, has been done by Pollack [17]. Coquand [11] has formalized a normalization function for the simply-typed λ-calculus using the proof editor ALF.

The rest of this article is structured as follows: in §2, we give the basic definitions of untyped λ-calculus. On top of this calculus, a system of simple types is introduced in §3. In §4 a definition of normal forms of λ-terms is given, which

serves as a basis for the proof of the normalization theorem presented in §5. An analysis of the program extracted from this proof is contained in §6.

2 Basic definitions

We start by introducing basic concepts such as terms and substitutions. The following definitions are due to Nipkow [14], who used them as a basis for a proof of the Church-Rosser property for β-reduction. They are reproduced here in order to make the exposition self-contained. λ-terms are modelled by the datatype *dB* using *de Bruijn indices*, which are encoded by natural numbers.

datatype $dB = Var\ nat \mid App\ dB\ dB \mid Abs\ dB$

We use $t \circ u$ as an infix notation for $App\ t\ u$. When substituting a term for a variable inside an abstraction, the indices of all free variables in the term have to be incremented. This is taken care of by the *lift* function

consts $lift :: dB \Rightarrow nat \Rightarrow dB$
primrec
$(Var\ i) \uparrow k = (if\ i < k\ then\ Var\ i\ else\ Var\ (i + 1))$
$(s \circ t) \uparrow k = s \uparrow k \circ t \uparrow k$
$(Abs\ s) \uparrow k = Abs\ (s \uparrow (k + 1))$

where $t \uparrow k$ is an infix notation for $lift\ t\ k$. Using $op \uparrow$, we can now define the substitution $t[s/k]$ of a term s for a variable k in a term t as follows:

consts $subst :: dB \Rightarrow dB \Rightarrow nat \Rightarrow dB$
primrec
$(Var\ i)[s/k] = (if\ k < i\ then\ Var\ (i - 1)\ else\ if\ i = k\ then\ s\ else\ Var\ i)$
$(t \circ u)[s/k] = t[s/k] \circ u[s/k]$
$(Abs\ t)[s/k] = Abs\ (t[s \uparrow 0\ /\ k{+}1])$

Since the substitution function will be used to specify β-reduction, it actually does not only substitute the term u for the variable i, but also decrements the indices of all other free variables by *1*, to compensate for the disappearance of abstractions during β-reduction. The definition of β-reduction, which is denoted by $s \rightarrow_\beta t$, is as usual:

consts $beta :: (dB \times dB)\ set$
inductive *beta*
intros
beta: $Abs\ s \circ t \rightarrow_\beta s[t/0]$
appL: $s \rightarrow_\beta t \Longrightarrow s \circ u \rightarrow_\beta t \circ u$
appR: $s \rightarrow_\beta t \Longrightarrow u \circ s \rightarrow_\beta u \circ t$
abs: $s \rightarrow_\beta t \Longrightarrow Abs\ s \rightarrow_\beta Abs\ t$

We also use $\rightarrow_\beta{}^*$ to denote the transitive closure of $\rightarrow_\beta$. The following congruence rules for $\rightarrow_\beta{}^*$ are occasionally useful in proofs:

lemma *rtrancl-beta-Abs*: $s \rightarrow_\beta{}^* s' \Longrightarrow Abs\ s \rightarrow_\beta{}^* Abs\ s'$
lemma *rtrancl-beta-AppL*: $s \rightarrow_\beta{}^* s' \Longrightarrow s \circ t \rightarrow_\beta{}^* s' \circ t$

lemma *rtrancl-beta-AppR*: $t \to_\beta^* t' \Longrightarrow s \circ t \to_\beta^* s \circ t'$
lemma *rtrancl-beta-App*: $s \to_\beta^* s' \Longrightarrow t \to_\beta^* t' \Longrightarrow s \circ t \to_\beta^* s' \circ t'$

We will also need the following theorems, asserting that $\to_\beta$ and $\to_\beta^*$ are *compatible* with lifting and substitution. The first two of these properties are called *substitutivity* in [12].

theorem *subst-preserves-beta*: $r \to_\beta s \Longrightarrow (\bigwedge t\ i.\ r[t/i] \to_\beta s[t/i])$
theorem *subst-preserves-beta'*: $r \to_\beta^* s \Longrightarrow r[t/i] \to_\beta^* s[t/i]$
theorem *lift-preserves-beta*: $r \to_\beta s \Longrightarrow (\bigwedge i.\ r \uparrow i \to_\beta s \uparrow i)$
theorem *lift-preserves-beta'*: $r \to_\beta^* s \Longrightarrow r \uparrow i \to_\beta^* s \uparrow i$
theorem *subst-preserves-beta2*: $\bigwedge r\ s\ i.\ r \to_\beta s \Longrightarrow t[r/i] \to_\beta^* t[s/i]$
theorem *subst-preserves-beta2'*: $r \to_\beta^* s \Longrightarrow t[r/i] \to_\beta^* t[s/i]$

In addition to the usual *binary* application operator $s \circ t$, it is often convenient to also have an n-ary application operator $t \circ\circ ts$ for applying a term t to a list of terms ts. To this end, we introduce the abbreviation

translations $t \circ\circ ts \rightleftharpoons foldl\ (op \circ)\ t\ ts$

The following equations, describing how lifting and substitution operate on such n-ary applications, are easily established by induction on the list ts:

lemma *lift-map*: $\bigwedge t.\ (t \circ\circ ts) \uparrow i = t \uparrow i \circ\circ map\ (\lambda t.\ t \uparrow i)\ ts$
lemma *subst-map*: $\bigwedge t.\ (t \circ\circ ts)[u/i] = t[u/i] \circ\circ map\ (\lambda t.\ t[u/i])\ ts$

3 Typed Lambda terms

In this section, we introduce the type system for simply-typed λ-calculus. The typing judgement usually depends on some environment (or context), assigning types to the free variables occurring in a term. Since variables are encoded using de Bruijn indices, it seems convenient to model environments as functions from natural numbers to types. In order to insert a type T into an environment e at a given position i, we define the function

constdefs
$shift :: (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$
$e\langle i{:}T\rangle \equiv \lambda j.\ if\ j < i\ then\ e\ j\ else\ if\ j = i\ then\ T\ else\ e\ (j - 1)$

where $e\langle i{:}T\rangle$ is syntactic sugar for *shift* $e\ i\ T$. The types of variables with indices less than i are left untouched, whereas the types of variables with indices greater than i are shifted one position up. Instead of working directly with the above definition, we will use the following characteristic theorems for *shift*.

lemma *shift-eq*: $i = j \Longrightarrow (e\langle i{:}T\rangle)\ j = T$
lemma *shift-gt*: $j < i \Longrightarrow (e\langle i{:}T\rangle)\ j = e\ j$
lemma *shift-lt*: $i < j \Longrightarrow (e\langle i{:}T\rangle)\ j = e\ (j - 1)$
lemma *shift-commute*: $e\langle i{:}U\rangle\langle 0{:}T\rangle = e\langle 0{:}T\rangle\langle Suc\ i{:}U\rangle$

Note that the above definition is actually a bit more polymorphic than necessary. We now come to the definition of types. In simply-typed λ-calculus, a type can either be an *atomic* type or a *function* type:

datatype *type* = *Atom nat* | *Fun type type*

In the sequel, we use $T \Rightarrow U$ as an infix notation for *Fun* T U. In analogy to the concept of an n-ary application, it is also useful to have an n-ary function type operator, which is characterized as follows:

translations $Ts \Rrightarrow T \rightleftharpoons \mathit{foldr}\ \mathit{Fun}\ Ts\ T$

Intuitively, $Ts \Rrightarrow T$ denotes the type of a function whose arguments have the types contained in the list Ts and whose result type is T. The definition of the typing judgement $e \vdash t : T$ is rather straightforward:

inductive *typing*
intros
VarT: $e\ x = T \Longrightarrow e \vdash \mathit{Var}\ x : T$
AbsT: $e\langle 0{:}T\rangle \vdash t : U \Longrightarrow e \vdash \mathit{Abs}\ t : (T \Rightarrow U)$
AppT: $e \vdash s : T \Rightarrow U \Longrightarrow e \vdash t : T \Longrightarrow e \vdash (s \circ t) : U$

In the typing rule for abstractions, the argument type T of the function is inserted at position 0 in the environment e when checking the type of the body t. The above typing judgement naturally extends to lists of terms. We write $e \Vdash ts : Ts$ to mean that the terms ts have types Ts. Formally, this extension of the typing judgement to lists of terms is defined as follows:

primrec
$(e \Vdash [] : Ts) = (Ts = [])$
$(e \Vdash (t \# ts) : Ts) =$
$(\mathit{case}\ Ts\ \mathit{of}\ [] \Rightarrow \mathit{False} \mid T \# Ts \Rightarrow e \vdash t : T \wedge e \Vdash ts : Ts)$

Using the above typing judgement for lists of terms, we can prove the following elimination and introduction rules for types of n-ary applications:

lemma *list-app-typeE*: $e \vdash t \circ\circ ts : T \Longrightarrow$
$(\bigwedge Ts.\ e \vdash t : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow P) \Longrightarrow P$
lemma *list-app-typeI*: $\bigwedge t\ T\ Ts.\ e \vdash t : Ts \Rrightarrow T \Longrightarrow$
$e \Vdash ts : Ts \Longrightarrow e \vdash t \circ\circ ts : T$

Before we come to the *subject reduction* theorem, which is the main result of this section, we need several additional results about lifting and substitution. The first two of these lemmas state that lifting preserves the type of a term:

lemma *lift-type*: $e \vdash t : T \Longrightarrow (\bigwedge i\ U.\ e\langle i{:}U\rangle \vdash t \uparrow i : T)$
lemma *lift-types*: $\bigwedge Ts.\ e \Vdash ts : Ts \Longrightarrow e\langle i{:}U\rangle \Vdash (\mathit{map}\ (\lambda t.\ t \uparrow i)\ ts) : Ts$

The first lemma is easily proved by induction on the typing derivation, whereas the second one, which is just a generalization of the first lemma to lists of terms, can be proved by induction on the list ts using the first result. The other two lemmas state that well-typed substitution preserves the type of terms:

lemma *subst-lemma*: $e \vdash t : T \Longrightarrow$
$(\bigwedge e'\ i\ U\ u.\ e' \vdash u : U \Longrightarrow e = e'\langle i{:}U\rangle \Longrightarrow e' \vdash t[u/i] : T)$
lemma *substs-lemma*: $\bigwedge Ts.\ e \vdash u : T \Longrightarrow$
$e\langle i{:}T\rangle \Vdash ts : Ts \Longrightarrow e \Vdash (\mathit{map}\ (\lambda t.\ t[u/i])\ ts) : Ts$

Again, the proof of the first lemma is by induction on the typing derivation, while the second one is proved by induction on *ts*. We are now ready to prove the *subject reduction* property, i.e. that $\rightarrow_\beta$ preserves the type of a term:

lemma *subject-reduction*: $e \vdash t : T \Longrightarrow (\bigwedge t'.\ t \rightarrow_\beta t' \Longrightarrow e \vdash t' : T)$

The proof is by induction on the typing derivation, where the cases for variables and abstractions are fairly trivial. The case dealing with applications $s \circ t$ can be proved using elimination on $s \circ t \rightarrow_\beta t'$ followed by an application of *subst-lemma*. This theorem easily extends to the transitive closure $\rightarrow_\beta{}^*$ of $\rightarrow_\beta$:

theorem *subject-reduction'*: $t \rightarrow_\beta{}^* t' \Longrightarrow e \vdash t : T \Longrightarrow e \vdash t' : T$

4 Terms in normal form

The definition which is central to the proof of weak normalization is, of course, that of a term in *normal form*. Intuitively, a term is in normal form, if it is either a variable applied to a list of terms in normal form, or an abstraction whose body is a term in normal form.

consts *NF* :: *dB set*
inductive *NF*
intros
AppN: *listall* $(\lambda t.\ t \in NF)\ ts \Longrightarrow Var\ x \circ\circ ts \in NF$
AbsN: $t \in NF \Longrightarrow Abs\ t \in NF$

In the above definition, we write *listall P xs* to denote that a predicate P holds for all elements in the list *xs*. We conclude this section by stating some properties of *NF*, which will be of particular importance for the main proof presented in the next section. As a trivial consequence of the above definition of normal forms, a term consisting of just a variable is in normal form.

lemma *Var-NF*: $Var\ n \in NF$

By substituting a variable i for a variable j in a normal term t, we obtain a term which is still in normal form:

lemma *subst-Var-NF*: $t \in NF \Longrightarrow (\bigwedge i\ j.\ t[Var\ i/j] \in NF)$

The above lemma is easily proved by induction on the derivation of $t \in NF$. If t is in normal form, the term $t \circ Var\ i$ possesses a normal form, too:

lemma *app-Var-NF*: $t \in NF \Longrightarrow \exists t'.\ t \circ Var\ i \rightarrow_\beta{}^* t' \wedge t' \in NF$

Again, this result can be proved by induction on the derivation of $t \in NF$, using the previous lemma *subst-Var-NF* in the abstraction case. Finally, lifting a normal term t again yields a normal term:

lemma *lift-NF*: $t \in NF \Longrightarrow (\bigwedge i.\ t \uparrow i \in NF)$

As usual, the proof is by induction on the derivation of $t \in NF$.

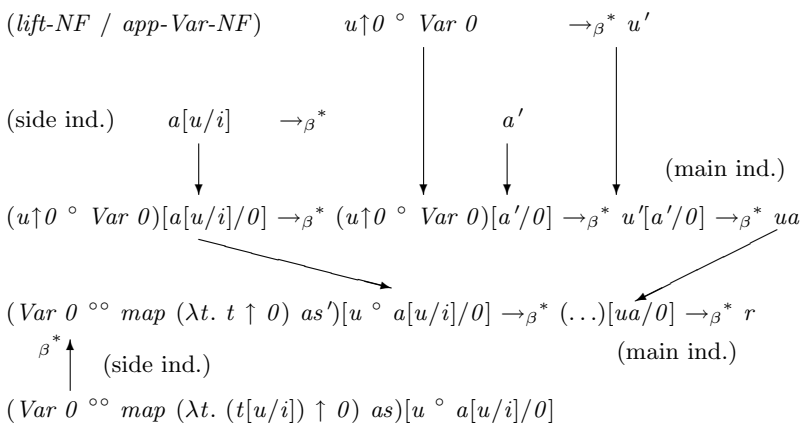

Fig. 1. Overview of proof for application case

5 Main theorems

We are now just one step away from our main result, the weak normalization theorem. Actually, the main difficulty is to prove a central lemma, from which weak normalization then follows by a relatively simple argument. The essence of this lemma can be summarized by the slogan "*well-typed substitution preserves the existence of normal forms*". More formally, if we have a well-typed term t in normal form containing a variable i of type U, then the term $t[u/i]$ obtained by substituting a term u in normal form of type U for the variable i can be reduced to a normal form t'. The proof of this statement, which we will now discuss in detail, is by main induction on the type U, followed by a side induction on the derivation of $t \in NF$. An interesting point to note is that the main induction hypothesis is used only in one case of the proof, whereas all the other cases are proved using the side induction hypothesis. In the following, we will give the proof of the lemma in the form of a commented script in the *Isar* proof language due to Markus Wenzel [19,15].

lemma *subst-type-NF*:
$\bigwedge t\ e\ T\ u\ i.\ t \in NF \Longrightarrow e\langle i{:}U\rangle \vdash t : T \Longrightarrow u \in NF \Longrightarrow e \vdash u : U \Longrightarrow$
$\exists t'.\ t[u/i] \to_\beta^*\ t' \wedge t' \in NF$

We start the proof by performing induction on the type U.

proof (*induct U*)
fix $T\ t$

We proceed by side induction on the derivation of $t \in NF$:

assume $t \in NF$

thus $\bigwedge e\ T'\ u\ i.\ e\langle i{:}T\rangle \vdash t : T' \Longrightarrow$
$u \in NF \Longrightarrow e \vdash u : T \Longrightarrow \exists t'.\ t[u/i] \rightarrow_\beta{}^*\ t' \wedge t' \in NF$
proof *induct*
fix $e\ T'\ u\ i$ **assume** *uNF*: $u \in NF$ **and** *uT*: $e \vdash u : T$
{
case (*AppN ts x e- T'- u- i-*)
assume $e\langle i{:}T\rangle \vdash Var\ x \circ\circ\ ts : T'$
then obtain *Us*
where *varT*: $e\langle i{:}T\rangle \vdash Var\ x : Us \Rrightarrow T'$
and *argsT*: $e\langle i{:}T\rangle \Vdash ts : Us$
by (*rule var-app-typesE*)

In the application case, we have to distinguish whether or not the variable x in the head of the term coincides with the variable i to be substituted.

from *nat-eq-dec* **show** $\exists t'.\ (Var\ x \circ\circ\ ts)[u/i] \rightarrow_\beta{}^*\ t' \wedge t' \in NF$
proof
assume *eq*: $x = i$
show *?thesis*

In this case, we do a case analysis on the argument list *ts*. If the argument list is empty, the claim follows trivially.

proof (*cases ts*)
case *Nil*
with *eq* **have** $(Var\ x \circ\circ\ [])[u/i] \rightarrow_\beta{}^*\ u$ **by** *simp*
with *Nil* **and** *uNF* **show** *?thesis* **by** *simp rules*
next
case (*Cons a as*)
. . .

The most difficult case of the proof is the one where the argument list is nonempty, i.e. the term on which the substitution is performed has the form $Var\ x \circ\circ\ (a \#\ as)$. The overall structure of the argument for this case is shown in Figure 1. Substitution and normalization will be performed in several steps. As a first step, we apply substitution and normalization to the tail *as* of the argument list. To this end, we prove the following intermediate statement:

from *AppN* **and** *Cons* **have** *listall ?SI as*
by *simp* (*rules dest*: *listall-conj2*)
with *lift-preserves-beta'* *lift-NF uNF uT argsT'*
have $\exists as'.\ \forall j.\ Var\ j \circ\circ\ map\ (\lambda t.\ (t[u/i]) \uparrow 0)\ as \rightarrow_\beta{}^*$
$Var\ j \circ\circ\ map\ (\lambda t.\ t \uparrow 0)\ as' \wedge$
$Var\ j \circ\circ\ map\ (\lambda t.\ t \uparrow 0)\ as' \in NF$ **by** (*rule norm-list*)
then obtain *as'* **where**
asred: $Var\ 0 \circ\circ\ map\ (\lambda t.\ (t[u/i]) \uparrow 0)\ as \rightarrow_\beta{}^*$
$Var\ 0 \circ\circ\ map\ (\lambda t.\ t \uparrow 0)\ as'$
and *asNF*: $Var\ 0 \circ\circ\ map\ (\lambda t.\ t \uparrow 0)\ as' \in NF$ **by** *rules*

The desired normal forms are guaranteed to exist due to the side induction hypothesis *listall ?SI as*. Since we later on want to substitute another term for the head variable *Var 0*, we also have to lift the argument terms, in order to avoid that they are affected by the substitution. In other words, the head variable has to be *new*. The above statement

is proved by "reverse induction" on *as*, i.e. elements are appended to the right of the list in the induction step. From the computational point of view, the existentially quantified variable *as'* acts as a kind of *accumulator* for the normalized terms. The proof relies on the fact that $\rightarrow_\beta{}^*$ is a congruence wrt. application and that $\rightarrow_\beta{}^*$ is compatible with lifting.
By using the side induction hypothesis one more time, we can also apply substitution and normalization to the head a of the argument list.

from *AppN* **and** *Cons* **have** *?SI a* **by** *simp*
with *argT* **and** *uNF* **and** *uT* **have** $\exists\, a'.\ a[u/i] \rightarrow_\beta{}^*\ a' \wedge a' \in NF$
  by *rules*
then obtain a' **where** *ared*: $a[u/i] \rightarrow_\beta{}^*\ a'$ **and** *aNF*: $a' \in NF$
  by *rules*

In order to show that the application of u to $a[u/i]$ has a normal form, too, we first note that the term u applied to a new variable again has a normal form. Since the argument type T'' of u is smaller than the type $T = T'' \Rightarrow Ts \Rrightarrow T'$ of i, we can use the main induction hypothesis, together with the previous result and compatibility of $\rightarrow_\beta{}^*$ with substitution, to show that also $u \circ a[u/i]$ has a normal form.

from *uNF* **have** $u \uparrow 0 \in NF$ **by** (*rule lift-NF*)
hence $\exists\, u'.\ u \uparrow 0 \circ Var\ 0 \rightarrow_\beta{}^*\ u' \wedge u' \in NF$ **by** (*rule app-Var-NF*)
then obtain u' **where** *ured*: $u \uparrow 0 \circ Var\ 0 \rightarrow_\beta{}^*\ u'$
  and *u'NF*: $u' \in NF$ **by** *rules*
from T **and** *u'NF* **have** $\exists\, ua.\ u'[a'/0] \rightarrow_\beta{}^*\ ua \wedge ua \in NF$
proof (*rule MI1*)
  ...
qed
then obtain *ua* **where** *uared*: $u'[a'/0] \rightarrow_\beta{}^*\ ua$
  and *uaNF*: $ua \in NF$ **by** *rules*
from *ared* **have** $(u \uparrow 0 \circ Var\ 0)[a[u/i]/0] \rightarrow_\beta{}^*\ (u \uparrow 0 \circ Var\ 0)[a'/0]$
  by (*rule subst-preserves-beta2'*)
also from *ured* **have** $(u \uparrow 0 \circ Var\ 0)[a'/0] \rightarrow_\beta{}^*\ u'[a'/0]$
  by (*rule subst-preserves-beta'*)
also note *uared*
finally have $(u \uparrow 0 \circ Var\ 0)[a[u/i]/0] \rightarrow_\beta{}^*\ ua$ **.**
hence *uared'*: $u \circ a[u/i] \rightarrow_\beta{}^*\ ua$ **by** *simp*

Finally, since the type $Ts \Rrightarrow T'$ of $u \circ a[u/i]$ is also smaller than $T = T'' \Rightarrow Ts \Rrightarrow T'$, we may again use the main induction hypothesis, together with the previous result, the above intermediate statement concerning the application of substitution and normalization to the argument list *as*, as well as compatibility of $\rightarrow_\beta{}^*$ with substitution, to show that also $u \circ a[u/i] \circ\circ map\ (\lambda t.\ t[u/i])\ as$ has a normal form.

from T **have**
  $\exists\, r.\ (Var\ 0 \circ\circ map\ (\lambda t.\ t \uparrow 0)\ as')[ua/0] \rightarrow_\beta{}^*\ r \wedge r \in NF$
proof (*rule MI2*)
  ...
qed
then obtain r
  where *rred*: $(Var\ 0 \circ\circ map\ (\lambda t.\ t \uparrow 0)\ as')[ua/0] \rightarrow_\beta{}^*\ r$
  and *rnf*: $r \in NF$ **by** *rules*

```
      from asred have
        (Var 0 °° map (λt. (t[u/i]) ↑ 0) as)[u ° a[u/i]/0] →β*
        (Var 0 °° map (λt. t ↑ 0) as')[u ° a[u/i]/0]
        by (rule subst-preserves-beta')
      also from ared' have
        (Var 0 °° map (λt. t ↑ 0) as')[u ° a[u/i]/0] →β*
        (Var 0 °° map (λt. t ↑ 0) as')[ua/0] by (rule subst-preserves-beta2')
      also note rred
      finally have
        (Var 0 °° map (λt. (t[u/i]) ↑ 0) as)[u ° a[u/i]/0] →β* r .
      with rnf Cons eq show ?thesis
        by (simp add: map-compose [symmetric] o-def) rules
    qed
```

This concludes the proof for the case where $x = i$.

```
    next
      assume neq: x ≠ i
      ...
    qed
```

The proof for this case is much easier than the previous one, although it is not completely trivial. As in the previous case, the side induction hypothesis has to be applied to all terms in the argument list *ts*. Again, the fact that $\rightarrow_\beta{}^*$ is a congruence wrt. application is required. This time, no lifting is involved, but the head variable may be decremented as a side effect of substitution (see §2) if $i < x$. This concludes the proof for the application case. The abstraction case follows by an easy application of the side induction hypothesis, using the fact that $\rightarrow_\beta{}^*$ is a congruence wrt. abstraction.

```
  next
    case (AbsN r e- T'- u- i-)
    assume absT: e⟨i:T⟩ ⊢ Abs r : T'
    then obtain R S where e⟨0:R⟩⟨Suc i:T⟩ ⊢ r : S
      by (rule abs-typeE) simp
    moreover have u ↑ 0 ∈ NF by (rule lift-NF)
    moreover have e⟨0:R⟩ ⊢ u ↑ 0 : T by (rule lift-type)
    ultimately have ∃ t'. r[u ↑ 0/Suc i] →β* t' ∧ t' ∈ NF by (rule AbsN)
    thus ∃ t'. Abs r[u/i] →β* t' ∧ t' ∈ NF
      by simp (rules intro: rtrancl-beta-Abs NF.AbsN)
  }
 qed
qed
```

Before we can embark on the proof of the main theorem of this section, stating that each well-typed λ-term has a normal form, there is another problem to solve. In the proof of the central lemma *subst-type-NF*, all the required typing information was easy to reconstruct even without inspecting the typing derivation, since the terms supplied as an input to the algorithm underlying the proof were already in normal form. This is no longer the case for the main theorem, of course, since its very purpose is the normalization of terms. As these terms do not contain any typing information themselves, this information has to be

```
theorem type-NF: assumes T: e ⊢R t : T
  shows ∃ t'. t →β* t' ∧ t' ∈ NF using T
proof induct
  case VarRT
  show ?case by (rules intro: Var-NF)
next
  case AbsRT
  thus ?case by (rules intro: rtrancl-beta-Abs AbsN)
next
  case (AppRT T U e s t)
  from AppRT obtain s' t' where
    sred: s →β* s' and sNF: s' ∈ NF
    and tred: t →β* t' and tNF: t' ∈ NF by rules
  have ∃ u. (Var 0 ∘ t' ↑ 0)[s'/0] →β* u ∧ u ∈ NF
  proof (rule subst-type-NF)
    have t' ↑ 0 ∈ NF by (rule lift-NF)
    hence listall (λt. t ∈ NF) [t' ↑ 0] by (rule listall-cons) (rule listall-nil)
    hence Var 0 ∘∘ [t' ↑ 0] ∈ NF by (rule AppN)
    thus Var 0 ∘ t' ↑ 0 ∈ NF by simp
    show e⟨0:T ⇒ U⟩ ⊢ Var 0 ∘ t' ↑ 0 : U
    proof (rule AppT)
      show e⟨0:T ⇒ U⟩ ⊢ Var 0 : T ⇒ U
        by (rule VarT) simp
      from tred have e ⊢ t' : T
        by (rule subject-reduction') (rule rtyping-imp-typing)
      thus e⟨0:T ⇒ U⟩ ⊢ t' ↑ 0 : T
        by (rule lift-type)
    qed
    from sred show e ⊢ s' : T ⇒ U
      by (rule subject-reduction') (rule rtyping-imp-typing)
  qed
  then obtain u where ured: s' ∘ t' →β* u and unf: u ∈ NF by simp rules
  from sred tred have s ∘ t →β* s' ∘ t' by (rule rtrancl-beta-App)
  hence s ∘ t →β* u using ured by (rule rtrancl-trans)
  with unf show ?case by rules
qed
```

Fig. 2. Proof of main theorem

obtained from the typing derivation. In order be able to formalize the main theorem, we therefore define a computationally relevant copy $e \vdash_R t : T$ of the typing judgement $e \vdash t : T$, where the subscript R stands for *R*elevant. The introduction rules characterizing this judgement are the same as for the original one. In order to plug the previous lemma into the proof of the main theorem, we will need the following rule, stating that the computationally relevant typing judgement implies the computationally irrelevant one:

```
lemma rtyping-imp-typing: e ⊢R t : T ⟹ e ⊢ t : T
```

This rule is easily proved by induction on $e \vdash_R t : T$. Note that the other direction would be provable as well, although, from the program extraction point of view, this would not make much sense, since there cannot be a program corresponding to the proof of a computationally relevant statement by induction on a computationally irrelevant statement. Note that instead of a computationally relevant typing judgement, we could as well have used a "Church-style" encoding of λ-terms, where variables in abstractions are decorated with types.
We are now ready to prove weak normalization, which will be done by induction on the typing derivation $e \vdash_R t : T$. All cases except for the application case are trivial. In order to normalize a term of the form $s \circ t$, we first use the induction hypothesis to compute the normal forms s' and t' of s and t, respectively. To show that also $s' \circ t'$ has a normal form, we first note that the application of a new variable to the term t' is in normal form, so the term obtained by substituting the term s' for this variable has a normal form according to lemma *subst-type-NF*. By transitivity, we can then put together the reduction sequences found in this way, to yield a normal form of $s \circ t$. The whole proof is shown in Figure 2.

6 Extracted programs

We conclude this case study with an analysis of the programs extracted from the proofs presented in the previous section. The programs corresponding to the proof of the main theorem *type-NF*, as well as the central lemma *subst-type-NF*, which performs substitution and normalization, is shown in Figure 3. The outer structure of the function *subst-type-NF* consists of two nested recursion combinators *type-induct-P* and *NFT-rec* corresponding to induction on types and the derivation of normal forms, respectively. The datatype representing the computational content of the inductive definition of normal forms is

$$\begin{array}{l}\mathsf{datatype}\ NFT\ = \\ \quad Dummy \\ \quad |\ AppN\ (dB\ list)\ nat\ (nat \Rightarrow NFT) \\ \quad |\ AbsN\ dB\ NFT\end{array}$$

The universal quantifier over list indices in the definition of the predicate *listall*, which is used in the first introduction rule of *NF* shown in §4, gives rise to the function type $nat \Rightarrow NFT$ in the list of argument types for the constructor *AppN* in the above datatype definition. Since the constructors *AppN* and *AbsN* both refer to the type *NFT* to be defined recursively, another *Dummy* constructor is required in order to ensure non-emptiness of the datatype. This datatype comes with a so-called *realizability predicate NFR*, which establishes a connection between terms in normal form and elements of the above datatype. This predicate is inductively defined by the rules

$$\forall\, i{<}length\ ts.\ (nfs\ i,\ ts\ !\ i) \in NFR \Longrightarrow (AppN\ ts\ x\ nfs,\ Var\ x\ {}^{\circ\circ}\ ts) \in NFR$$
$$(nf,\ t) \in NFR \Longrightarrow (AbsN\ t\ nf,\ Abs\ t) \in NFR$$

```
subst-type-NF ≡
λx xa xb xc xd xe H Ha.
  type-induct-P xc
   (λx H2 H2a xa xb xc xd xe H.
      NFT-rec arbitrary
       (λts xa xaa r xb xc xd xe H.
          var-app-typesE-P (xb⟨xe:x⟩) xa ts
           (λUs. case nat-eq-dec xa xe of
                   Left ⇒
                    case ts of [] ⇒ (xd, H)
                    | a # list ⇒
                       case Us of [] ⇒ arbitrary
                       | T'' # Ts ⇒
                          let (x, y) =
                                norm-list (λt. t ↑ 0) xd xb xe list Ts
                                 (λt. lift-NF 0) H
                                 (listall-conj2-P-Q (λi. (xaa (Suc i), r (Suc i))));
                              (xa, ya) = snd (xaa 0, r 0) xb T'' xd xe H;
                              (xd, yb) = app-Var-NF 0 (lift-NF 0 H);
                              (xa, ya) = H2 T'' (Ts ⇛ xc) xd xb (Ts ⇛ xc) xa 0 yb ya
                          in H2a T'' (Ts ⇛ xc) (Var 0 °° map (λt. t ↑ 0) x) xb xc xa
                              0 (y 0) ya
                   | Right ⇒
                      let (x, y) =
                            norm-list (λt. t) xd xb xe ts Us (λx H. H) H
                             (listall-conj2-P-Q (λz. (xaa z, r z)))
                      in case nat-le-dec xe xa of
                           Left ⇒ (Var (xa − Suc 0) °° x, y (xa − Suc 0))
                           | Right ⇒ (Var xa °° x, y xa)))
       (λt x r xa xb xc xd H.
          abs-typeE-P xb
           (λU V. let (x, y) = r (xa⟨0:U⟩) V (xc ↑ 0) (Suc xd) (lift-NF 0 H)
                  in (Abs x, AbsN x y)))
        H xb xc xd xe)
    x xa xd xe xb H Ha

type-NF ≡
λH. rtypingT-rec (λe x T. (Var x, Var-NF x))
     (λe T t U x r. let (x, y) = r in (Abs x, AbsN x y))
     (λe s T U t x xa r ra.
        let (x, y) = r; (xa, ya) = ra
        in subst-type-NF (Var 0 ° xa ↑ 0) e 0 (T ⇒ U) U x
             (AppN [xa ↑ 0] 0 (listall-cons-P (lift-NF 0 ya) listall-nil-P)) y)
     H
```

Fig. 3. Programs extracted from main theorem and *subst-type-NF*

which are derived in a canonical way from the introduction rules of the predicate *NF*. Intuitively, we can think of $(nf,\ t) \in NFR$ to mean that nf is a witness of the fact that the term t is in normal form. Note that $(nf,\ t) \in NFR \Longrightarrow t \in NF$, which is easily proved by induction on the derivation of $(nf,\ t) \in NFR$.
The recursion combinator *NFT-rec* occurring in the program shown in Figure 3 has three functions as arguments, corresponding to the constructors of the above datatype. The first function corresponds to the *Dummy* constructor. Since this constructor may never occur, we may supply an *arbitrary* function as an argument, which, when generating executable code, may be implemented by a function raising an exception on invocation. The second function corresponds to the application case of the proof. It contains a case distinction (using function *nat-eq-dec*) on whether the variable *xa* coincides with the variable *xe*[1]. The first case (labelled with *Left*), which is the more difficult one, contains another case distinction on the structure of the argument list. The second case (labelled with *Right*) is the easier one. It contains another case distinction (using function *nat-le-dec*) on whether $xe < xa$. In the "*Left*" case, the variable in the head of the term is decremented, whereas it remains unchanged in the "*Right*" case. In both the case for $xa = xe$ and $xa \neq xe$ the function *norm-list* is used to apply the normalization function to a list of terms. The last lines of the program shown in Figure 3 contain the relatively trivial program corresponding to the proof for the abstraction case. The correctness theorem corresponding to the program *subst-type-NF* is

$$\begin{array}{l}
\bigwedge x.\ (x,\ t) \in NFR \Longrightarrow \\
\quad e\langle i{:}U\rangle \vdash t : T \Longrightarrow \\
\quad (\bigwedge xa.\ (xa,\ u) \in NFR \Longrightarrow \\
\qquad e \vdash u : U \Longrightarrow \\
\qquad t[u/i] \rightarrow_\beta{}^* fst\ (subst\text{-}type\text{-}NF\ t\ e\ i\ U\ T\ u\ x\ xa) \wedge \\
\qquad (snd\ (subst\text{-}type\text{-}NF\ t\ e\ i\ U\ T\ u\ x\ xa), \\
\qquad\ fst\ (subst\text{-}type\text{-}NF\ t\ e\ i\ U\ T\ u\ x\ xa)) \\
\qquad \in NFR)
\end{array}$$

The function *type-NF* is defined by recursion on the datatype

$$\begin{array}{l}
\mathsf{datatype}\ rtypingT\ = \\
\quad VarRT\ (nat \Rightarrow type)\ nat\ type \\
\ \mid\ AbsRT\ (nat \Rightarrow type)\ type\ dB\ type\ rtypingT \\
\ \mid\ AppRT\ (nat \Rightarrow type)\ dB\ type\ type\ dB\ rtypingT\ rtypingT
\end{array}$$

representing the computational content of the typing derivation. The correctness statement for the main function *type-NF* is

$$\begin{array}{l}
\bigwedge ty.\ (ty,\ e,\ t,\ T) \in rtypingR \Longrightarrow \\
\quad t \rightarrow_\beta{}^* fst\ (type\text{-}NF\ ty) \wedge (snd\ (type\text{-}NF\ ty),\ fst\ (type\text{-}NF\ ty)) \in NFR
\end{array}$$

[1] Due to the numerous transformations, which are performed on the proof before extraction, variable names in the extracted programs may often differ from those in the original Isar proof document.

where *rtypingR* is the realizability predicate corresponding to the computationally relevant version of the typing judgement. In analogy to *NFR*, we can think of $(ty,\ e,\ t,\ T) \in rtypingR$ to mean that ty is a witness of the fact that t has type T in environment e. The reduction relation $\rightarrow_\beta{}^*$ has been chosen to have no computational content, since we are only interested in the normal form of a term, and not the actual *reduction sequence* leading to it.
Compared to the programs which are extracted "manually" by Matthes and Joachimski [12, §2.3], the automatically extracted programs presented in this section may look a bit more complicated and harder to read. This is due to the fact that, although (according to the proof) the main recursion in the program given by Matthes and Joachimski should be over types, no type information is mentioned in the extracted program at all. Moreover, the program looks as if it were defined by recursion over terms, whereas, strictly speaking, it should involve recursion over the datatype *NFT* representing the computational content of the inductive characterization of normal forms.
In order to test the performace of the extracted normalization function, we compile it to ML and use it to compute multiplication on *Church numerals*:

$$2 =_{def} (\lambda f.\ \lambda x.\ f\ (f\ x))$$
$$\star =_{def} (\lambda m.\ \lambda n.\ \lambda f.\ m\ (n\ f))$$

The time required for computing the normal form of $x \star 2$ on a Pentium III with 1 GHz is as follows:

x	2	4	8	16	32	64	128	256	512
runtime $[s]$	0.002	0.016	0.012	0.036	0.096	0.430	2.615	27.786	364.193

7 Conclusion

In this article, we have presented a fully-formalized, readable proof of a fairly complex result for simply-typed λ-calculus. All the definitions, theorems and proofs shown in this article have been directly generated from the Isabelle proof scripts. The whole formalization is quite compact. It consists of three main modules, one containing basic results about untyped λ-calculus, in particular the definition and properties of β-reduction, another one containing results about the type system, and one containing properties of normal forms, as well as proofs of the main theorems. Each of these modules takes up about 400 lines of Isabelle code. Using Isabelle's built-in code generator, it is possible to generate executable ML code from the extracted normalization function presented in §6, which performs reasonably well on medium-size λ-terms.

References

1. T. Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, University of Edinburgh, November 1993.

2. T. Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference (TLCA '93)*, volume 664 of *LNCS*, pages 13–28. Springer-Verlag, 1993.
3. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The POPLMARK Challenge. In T. Melham and J. Hurd, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2005*, LNCS. Springer-Verlag, 2005.
4. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, Nov. 1999.
5. B. Barras and B. Werner. Coq in Coq. To appear in Journal of Automated Reasoning.
6. B. Barras et al. The Coq proof assistant reference manual – version 7.2. Technical Report 0255, INRIA, February 2002.
7. H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II: Systems and Implementation Techniques of *Applied Logic Series*, pages 41–71. Kluwer Academic Publishers, Dordrecht, 1998.
8. U. Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference (TLCA '93)*, pages 91–106, Berlin, Germany, 1993. Springer-Verlag.
9. S. Berghofer. Program Extraction in simply-typed Higher Order Logic. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*. Springer-Verlag, 2003.
10. S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Institut für Informatik, TU München, 2003.
11. C. Coquand. From semantics to rules: A machine assisted analysis. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers*, volume 832 of *LNCS*, pages 91–105. Springer-Verlag, 1993.
12. F. Joachimski and R. Matthes. Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and Gödel's T. *Archive for Mathematical Logic*, 42(1):59–87, 2003.
13. Z. Luo and R. Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.
14. T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.
15. T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 259–278. Springer-Verlag, 2003.
16. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
17. R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.
18. W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.
19. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, TU München, 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.