Normalization for the Simply-Typed Lambda-Calculus in Twelf

Andreas Abel[1]

Institut für Informatik, Ludwig-Maximilians-Universität
Oettingenstr. 67, D-80538 München, Germany

Abstract

Normalization for the simply-typed λ-calculus is proven in Twelf, an implementation of the Edinburgh Logical Framework. Since due to proof-theoretical restrictions Twelf Tait's computability method does not seem to be directly usable, a syntactical proof is adapted and formalized instead. In this case study, some boundaries of Twelf current capabilities are touched and discussed.

Keywords: Edinburgh Logical Framework, HOAS, Mechanized Proof, Normalization, Twelf

1 Introduction

Twelf is an implementation of the Edinburgh Logical Framework which supports reasoning in full higher-order abstract syntax (HOAS); therefore it is an ideal candidate for reasoning comfortably about properties of prototypical programming languages with binding. Previous work has focused on properties like subject reduction, confluence, compiler correctness. Even cut elimination for various sequent calculi has been proven successfully. But until recently, there were no formalized proofs of normalization [2] in Twelf. The reason might be that normalization is typically proven by Tait's method, which cannot be applied directly in Twelf. This work explains why Tait's method is at least not directly applicable and provides a syntactical proof for the simply-typed lambda-calculus.

---

[2] There have been normalization proofs in logical frameworks with inductive definitions, for instance, Altenkirch's proof of strong normalization for System F in LEGO [2]. Since HOAS is not available in a framework like LEGO, he represents terms using de Bruijn indices.

$$\begin{array}{lll}
\mathbb{K} & ::= \mathsf{type} & \text{kind of types} \\
& | \; \{\mathbb{X}{:}\mathbb{A}\}\mathbb{K} & \text{dependent function kind} \\[4pt]
\mathbb{A} & ::= \mathbb{F} \, \mathbb{M}_1 \dots \mathbb{M}_n & \text{base type} \\
& | \; \{\mathbb{X}{:}\mathbb{A}\}\mathbb{A} & \text{dependent function type} \\
& | \; \mathbb{A} \to \mathbb{A} & \text{non-dependent function type} \\[4pt]
\mathbb{M} & ::= \mathbb{C} & \text{term constant} \\
& | \; \mathbb{X} & \text{term variable} \\
& | \; [\mathbb{X}{:}\mathbb{A}]\mathbb{M} & \text{term abstraction} \\
& | \; \mathbb{M}\,\mathbb{M} & \text{term application}
\end{array}$$

Fig. 1. Syntactic classes of LF.

2 Twelf

The Edinburgh Logical Framework (LF [3]) [6,7] is a dependently-typed lambda-calculus with type families and $\beta\eta$-equality, but neither polymorphism, inductive data types nor recursion. Expressions are divided into three syntactic classes: kinds, types and terms, generated by the grammar in Fig. 1. Herein, the meta variable $\mathbb{X}$ ranges over a countably infinite set of variable identifiers, while $\mathbb{F}$ resp. $\mathbb{C}$ range over type-family resp. term constants provided in a signature Σ. Note that neither a type nor a kind can depend on a type; consequently, abstraction is missing on the type level [11, p. 1124].

The framework comes with judgements for typing, $\mathbb{M} : \mathbb{A}$, kinding, $\mathbb{A} : \mathbb{K}$, and wellformedness of kinds, $\mathbb{K}$ *kind*, plus $\beta\eta$-equality on for terms, types, and kinds [7]. An object theory can be described in the framework by providing a suitable signature Σ which adds kinded type family constants $\mathbb{F} : \mathbb{K}$ and typed term constants $\mathbb{C} : \mathbb{A}$.

Twelf [12] is an implementation of LF whose most fundamental task is to check typing (and kinding) of a user given signature Σ, usually provided as a set of ASCII files. Symbols reserved for the framework are the following.

$$: \; . \; (\;) \; [\;] \; \{ \; \} \; \text{-> type}$$

All others can be used to denote entities in the object theories. In the remainder of this section, we show how to represent the simply-typed λ-calculus with weak head reduction, as specified in Fig. 2, in Twelf.

2.1 Representation of Syntactic Objects

Untyped lambda terms t can be represented by one type family constant `tm` and two term constants:

```
tm    : type.
lam   : (tm -> tm) -> tm.
app   : tm -> tm -> tm.
```

---

[3] This is not to be confused with Martin-Löf's framework for dependent type theory, which is also abbreviated by LF.

2

Syntax.

$$
\begin{array}{lll}
r, s, t, u & ::= & x \mid \lambda x.t \mid r\,s \qquad \text{untyped terms} \\
A, B, C & ::= & * \mid A \to B \qquad \text{simple types} \\
\Gamma & ::= & \diamond \mid \Gamma,\, x{:}A \qquad \text{contexts (all variables distinct)}
\end{array}
$$

Type assignment $\Gamma \vdash t : A$.

$$
\frac{(x{:}A) \in \Gamma}{\Gamma \vdash x : A}\ \texttt{of\_var}
$$

$$
\frac{\Gamma,\, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ \texttt{of\_lam}
\qquad
\frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\,s : B}\ \texttt{of\_app}
$$

Weak head reduction $t \longrightarrow_{\mathsf{w}} t'$.

$$
\frac{}{(\lambda x.t)\,s \longrightarrow_{\mathsf{w}} [s/x]t}\ \texttt{beta}
\qquad
\frac{r \longrightarrow_{\mathsf{w}} r'}{r\,s \longrightarrow_{\mathsf{w}} r'\,s}\ \texttt{appl}
$$

Fig. 2. Simply-typed λ-calculus and weak head reduction.

The lack of a construct for variables is due to the use of HOAS: object variables are represented by variables of the framework, e. g., in the code for the twice function:

```
twice = lam [f:tm] lam [x:tm] app f (app f x).
```

A more detailed explanation of higher-order encodings has been given by Schürmann [17, p. 20ff]. Simple types A can be generated from a nullary constant `*` for some base type and a binary constant `=>`, used infix, for function type formation.

```
ty      : type.
*       : ty.
=>      : ty -> ty -> ty.                    %infix right 10 =>.
```

2.2   Representation of Judgements and Relations

Type assignment for untyped terms, $\Gamma \vdash t : A$, can be represented by two constants as well: one for function introduction and one for function elimination. Note that in Twelf syntax, the types of new constants may contain free variables (capital letters), which are regarded as universally quantified on the outside.

```
of      : tm -> ty -> type.                  %infix none 1 of.
of_lam : ({x:tm} x of A -> (T x) of B)
            -> (lam [x:tm] T x) of (A => B).
of_app : R of (A => B) -> S of A -> (app R S) of B.
```

Again, there is no separate rule for the typing of variables, instead it is part of the rule for abstraction. The premise of rule `of_lam` is to be read as:

Consider a temporary extension of the signature by a fresh constant `x:tm` and assume `x of A`. Then `(T x) of B` holds.

3

This adds a *dynamical* typing rule `x of A` for each new variable `x` instead of inserting a typing hypothesis $x : A$ into the typing context Γ. Hence, we do not explicitly encode Γ, but let the framework handle the typing hypotheses. The shape of such dynamical rules can be declared in Twelf by a `%worlds` statement. In the case of typing, this would be the following:

```
%block of_var : some {A:ty} block {x:tm}{p:x of A}.
%worlds (of_var) (of _ _).
```

This states that in `of`-derivations, only assumptions blocks that are of the form `x:tm,p:x of A` for some `A:ty` enter the global Twelf context. Worlds declarations are required for coverage checking (see below).

Similar to the typing relation, we can represent weak head reduction $t \longrightarrow_{\mathsf{w}} t'$, which eliminates the head (resp. key) redex in term t but does not step under a binding.

```
-->w  : tm -> tm -> type.              %infix none 1 -->w.
beta  : app (lam T) S -->w T S.
appl  : R -->w R' -> app R S -->w app R' S.
```

One advantage of HOAS is that substitution does not have to be defined, but can be inherited from the framework. Since in rule `beta`, term `T : tm -> tm` is λ-function, substitution $[u/y]t$ is simply expressed as application `T U`.

Lemma 2.1 *If* $t \longrightarrow_{\mathsf{w}} t'$ *then* $[u/y]t \longrightarrow_{\mathsf{w}} [u/y]t'$.

Proof. By induction on the derivation of $t \longrightarrow_{\mathsf{w}} t'$.

- Case $(\lambda x.t)\, s \longrightarrow_{\mathsf{w}} [s/x]t$. W.l.o.g. $x \neq y$ and x not free in u. Then,

$$
\begin{aligned}
[u/y]((\lambda x.t)\, s) &= (\lambda x.[u/y]t)\, [u/y]s \\
&\longrightarrow_{\mathsf{w}} [[u/y]s/x][u/y]t = [u/y][s/x]t.
\end{aligned}
$$

- Case $r\, s \longrightarrow_{\mathsf{w}} r'\, s$ with $r \longrightarrow_{\mathsf{w}} r'$. By ind. hyp., $[u/y]r \longrightarrow_{\mathsf{w}} [u/y]r'$. Hence,

$$
\begin{aligned}
[u/y](r\, s) &= ([u/y]r)\, ([u/y]s) \\
&\longrightarrow_{\mathsf{w}} ([u/y]r')\, ([u/y]s) = [u/y](r'\, s)
\end{aligned}
$$

$\square$

Fig. 3. Weak head reduction is closed under substitution.

2.3 Representation of Theorems and Proofs

Fig. 3 shows the first lemma of our object theory. How do we represent it? Twelf's internal logic is constructive, therefore the lemma must be interpreted constructively: Given a derivation $\mathcal{P}$ of $t \longrightarrow_{\mathsf{w}} t'$ and a term u, we can construct a derivation $\mathcal{P}'$ of $[u/y]t \longrightarrow_{\mathsf{w}} [u/y]t'$. In type theories with inductive types and recursion, like Agda, Coq [8] and LEGO [14], the lemma would be represented as a recursive

function of the dependent type

$$\Pi t : tm. \, \Pi t' : tm. \, \Pi \mathcal{P} : t \longrightarrow_{\mathsf{w}} t'. \, \Pi u : tm. \, \Pi y : var. \, [u/y]t \longrightarrow_{\mathsf{w}} [u/y]t'.$$

In Twelf, however, with no recursive functions at hand, the lemma is represented as a relation between input and output derivations, and, thus, via the propositions-as-types paradigm, as just another type family.

```
subst_red : {U:tm} ({y:tm} T y -->w T' y)
                         -> T U -->w T' U -> type.
%mode subst_red +U +P -P'.
```

The `%mode` statement marks the first two arguments of type family `subst_red` as inputs (+) and the third as output (−). Thus, the lemma is a functional relation, and its proof is a logic program with two clauses, one for each case in the proof.

```
subst_red_beta: subst_red U ([y] beta) beta.
subst_red_appl: subst_red U ([y] appl (P y)) (appl P')
                <- subst_red U P P'.
%terminates P (subst_red _ P _).
```

The base case of the induction is given by the constant `subst_red_beta`, and the step case, which appeals to the induction hypothesis, by `subst_red_appl`. The *types* of these constants are the actual program and correspond to PROLOG clauses. Note that in the second type a reversed arrow "`<-`", which resembles PROLOG's "`:-`", has been used to encourage an operational reading:

Substitution in a derivation which can be split into last rule `appl` and remainder P — P may mention `y` — results in a derivation P' extended by an application of rule `appl`. Herein, P' is constructed from P recursively.

Since it is a logic program, we can even "execute" the lemma. Execution in Twelf is search: Given a type with free variables, find an inhabitant of the type and solutions for the free variables. For example:

```
P : {y} app (app (lam [x] x) y) y -->w app y y
  = [y] appl beta.
%define P' = X
%solve K : subst_red (lam [z] z) P X.
```

This defines a 2-rule derivation P which witnesses that $(\lambda x.x) \, y \, y \longrightarrow_{\mathsf{w}} y \, y$. The `%solve` statement asks Twelf for a derivation P' which arises from P by substituting $\lambda z.z$ for y, according to the lemma. The answer is:

```
P' : app (app (lam [x] x) (lam [z] z)) (lam [z] z)
        -->w app (lam [z] z) (lam [z] z)
   = appl beta.
K  : subst_red (lam [z] z) P (appl beta)
   = subst_red_appl subst_red_beta.
```

Since the value of P' equals P, the shape of the derivation has not changed, only its result: the type of P'. The value of K gives an execution trace of logic program `subst_red`: First, clause `subst_red_appl` has fired, then clause `subst_red_beta`

has concluded the search.

A logic program in Twelf corresponds to a partial function from inputs to outputs as specified by the mode declaration. Since only total functions correspond to valid inductive proofs we must ensure that the defined function *terminates* on all inputs and *covers* all possible cases. Both properties cannot be shown *within* the framework, e.g., we cannot give a proof that `subst_red` is terminating. Instead, totality of a function needs external reasoning and can be ensured by built-in tactics.

Rohwedder and Pfenning [15] and Pientka [13] contributed a termination checker which is invoked by the `%terminates` pragma. In the case of `subst_red`, the second argument P decreases structurally in each recursive call. Case coverage is ensured by an algorithm by Pfenning and Schürmann [18]. Both termination and coverage checking are necessarily incomplete. But all proofs developed in the remainder of this article pass the totality check.

3   A Formalized Proof of Weak Normalization

In this section, we present a combinatorial proof of weak normalization for the simply-typed lambda-calculus. It is similar to the textbook proof in Girard, Lafont and Taylor [4, Ch. 4], but we avoid reasoning with numbers altogether. In fact, we follow closely the very syntactical presentation of Joachimski and Matthes [9], which has also been implemented in Isabelle/Isar by Nipkow and Berghofer [3]. The main obstacle to a direct formalization in Twelf is the use of a vector notation for terms by Joachimski and Matthes, which allows them to reason on a high level in some cases. In this section, we will see a "de-vectorized" version of their proof which can be outlined as follows:

 (i) Define an inductive relation $t \Uparrow A$.
 (ii) Prove that for every term $t : A$ the relation $t \Uparrow A$ holds.
(iii) Show that every term in the relation is weakly normalizing.

3.1   Inductive Characterization of Weak Normalization

Inductive characterizations of normalization go back to Goguen [5] and van Raamsdonk and Severi [20] and seem to have been independently discovered by Valentini [19]. However all these definitions use vectors of arguments for λ-terms. To make such a definition Twelf-digestible, we have found a vector-free formulation. We introduce a relation $\Gamma \vdash t \Uparrow A$ which stipulates that t is weakly normalizing of type A, and an auxiliary relation $\Gamma \vdash t \Downarrow^x A$ which additionally claims that $t = x\,\boldsymbol{s}$ for

some sequence of normalizing terms $\boldsymbol{s}$, i.e., t is neutral resp. head-redex free.

$$\frac{(x\,{:}\,A)\in\Gamma}{\Gamma\vdash x\Downarrow^x A}\;\texttt{wne\_var}\qquad\frac{\Gamma\vdash r\Downarrow^x A\to B\qquad\Gamma\vdash s\Uparrow A}{\Gamma\vdash r\,s\Downarrow^x B}\;\texttt{wne\_app}$$

$$\frac{\Gamma\vdash r\Downarrow^x A}{\Gamma\vdash r\Uparrow A}\;\texttt{wn\_ne}\qquad\frac{\Gamma,x\,{:}\,A\vdash t\Uparrow B}{\Gamma\vdash\lambda x.t\Uparrow A\to B}\;\texttt{wn\_lam}$$

$$\frac{r\longrightarrow_{\mathsf{w}} r'\qquad\Gamma\vdash r'\Uparrow A}{\Gamma\vdash r\Uparrow A}\;\texttt{wn\_exp}$$

The Twelf representation is similar to the typing relation: Again, Γ and the hypothesis rule are indirectly represented in rule $\texttt{wn\_lam}$ and the worlds declaration.

```
var     : tm -> type.
wne     : tm -> ty -> var X -> type.
wn      : tm -> ty -> type.

wne_app : wne R (A => B) V -> wn S A -> wne (app R S) B V.
wn_ne   : wne R A V -> wn R A.
wn_lam  : ({x:tm} {v:var x} wne x A v -> wn (T x) B)
             -> wn (lam T) (A => B).
wn_exp  : R -->w R' -> wn R' A -> wn R A.

%block wne_var : some{A:ty}block {x:tm}{v:var x}{d:wne x A v}.
%worlds (wne_var) (wne _ _ _) (wn _ _).
```

The judgement $\texttt{var X}$ states that $\texttt{X}$ is a variable of type $\texttt{tm}$, since there are rules to introduce $\texttt{var}$. This formalization of *term variable* was suggested to me by Chung-Chieh Shan and enables coverage checking.

3.2 Closure under Application and Substitution

To show that each typed term $t : A$ is in the relation $t\Uparrow A$, we will proceed by induction on the typing derivation. Difficult is the case for an application of the form $(\lambda x.r)\,s$. It can only be shown to be in the relation by rule $\texttt{wn\_exp}$, which requires us to prove that $[s/x]r$ is in the relation. If x is head variable of r, substitution might create new redexes. In this case, however, we can argue that the type of r is a smaller type than the one of s. These preliminary thoughts lead to the following lemma.

Lemma 3.1 (Application and Substitution) *Let $\mathcal{D}::\Gamma\vdash s\Uparrow A$.*

(i) *If $\mathcal{E}::\Gamma\vdash r\Uparrow A\to C$ then $\Gamma\vdash r\,s\Uparrow C$.*

(ii) *If $\mathcal{E}::\Gamma,x\,{:}\,A\vdash t\Uparrow C$, then $\Gamma\vdash[s/x]t\Uparrow C$.*

(iii) *If $\mathcal{E}::\Gamma,x\,{:}\,A\vdash t\Downarrow^x C$, then $\Gamma\vdash[s/x]t\Uparrow C$ and C is a part of A.*

(iv) *If $\mathcal{E}::\Gamma,x\,{:}\,A\vdash t\Downarrow^y C$ with $x\neq y$, then $\Gamma\vdash[s/x]t\Downarrow^y C$.*

In Twelf, the lemma is represented by four type families. The invariant that C is a subexpression of A will be expressed via a $\texttt{\%reduces}$ statement later, which

makes is necessary to make type C an explicit argument to type family `subst_x`.

```
app_wn   : {A:ty} wn S A
                  -> wn R (A => C)  -> wn (app R S) C -> type.

subst_wn : {A:ty} wn S A ->         ({x:tm} {v:var x}
       wne x A v -> wn (T x) C)    -> wn (T S) C      -> type.

subst_x  : {A:ty} wn S A -> {C:ty} ({x:tm} {v:var x}
       wne x A v -> wne (T x) C v) -> wn (T S) C      -> type.

subst_y  : {A:ty} wn S A ->         ({x:tm} {v:var x}
       wne x A v -> wne (T x) C Y) -> wne (T S) C Y  -> type.

%mode app_wn    +A +D    +E -F.
%mode subst_wn +A +D     +E -F.
%mode subst_x  +A +D +C +E -F.
%mode subst_y  +A +D     +E -F.
```

Proof of Lemma 3.1 Simultaneously by main induction on type A and side induction on the derivation $\mathcal{E}$. (By this we mean lexicographic induction on $(A, \mathcal{E})$.)

(i) Show $\Gamma \vdash r\, s \Uparrow C$. If the last rule of $\mathcal{E}$ was `wn_ne`, hence, r is neutral, then $r\, s$ is also neutral by rule `wne_app`, thus, it is in the relation. If the last rule was `wn_exp`, we can apply the side ind. hyp. The interesting case is $r = \lambda x.t$ and

$$\frac{\Gamma, x : A \vdash t \Uparrow C}{\Gamma \vdash \lambda x.t \Uparrow A \to C} \text{ wn\_lam.}$$

Here, we proceed by side ind. hyp. ii. Here is the corresponding Twelf code (we repeat the type of `app_wn`).

```
app_wn   : {A:ty} wn S A
                  -> wn R (A => C)  -> wn (app R S) C -> type.

app_wn_ne   : app_wn A D (wn_ne E) (wn_ne (wne_app E D)).
app_wn_exp  : app_wn A D (wn_exp P E) (wn_exp (appl P) F)
                 <- app_wn A D E F.
app_wn_lam  : app_wn A D (wn_lam E) (wn_exp beta F)
                 <- subst_wn A D E F.
```

(ii) Show $\Gamma \vdash [s/x]t \Uparrow C$ for $\Gamma, x : A \vdash t \Uparrow C$. If t is not neutral, we conclude by ind. hyp. and possibly Lemma 2.1. Otherwise, we distinguish on the head variable of t: is it x, then we proceed by side ind. hyp. iii, otherwise by side ind. hyp. iv.

```
subst_wn : {A:ty} wn S A -> ({x:tm} {v:var x} wne x A v
   -> wn (T x) C) -> wn (T S) C -> type.

subst_wn_lam: subst_wn A D ([x][vx][dx]
       wn_lam ([y][vy][dy] E y vy dy x vx dx)) (wn_lam F)
```

8

```
      <- {y}{vy}{dy} subst_wn A D (E y vy dy) (F y vy dy).

  subst_wn_exp: subst_wn A (D : wn S A) ([x][v][dx]
         wn_exp (P x) (E x v dx)) (wn_exp P' E')
    <- subst_wn A D E E'
    <- subst_red S P P'.

  subst_wn_x  : subst_wn A D ([x][v][dx]
         (wn_ne (E x v dx) : wn (T x) C)) F
    <- subst_x A D C E F.

  subst_wn_y  : subst_wn A D ([x][v][dx]
         wn_ne (E x v dx)) (wn_ne F)
    <- subst_y A D E F.
```

(iii) Show $\Gamma \vdash [s/x]t \Uparrow C$ for $\Gamma' \vdash t \Downarrow^x C$ with $\Gamma' := \Gamma, x : A$. In case $t = x$, the type C is trivially a part of $A = C$ and we conclude by assumption $\Gamma \vdash s \Uparrow C$. Otherwise, $t = r\,u$ and the last rule in $\mathcal{E}$ was

$$\frac{\Gamma' \vdash r \Downarrow^x B \to C \qquad \Gamma' \vdash u \Uparrow B}{\Gamma' \vdash r\,u \Downarrow^x C} \quad \text{wne\_app}.$$

By side ind. hyp. iii we know that $B \to C$ is a part of A and $\Gamma \vdash r' \Uparrow B \to C$ where $r' := [s/x]r$. Similarly $\Gamma \vdash u' \Uparrow B$ for $u' := [s/x]u$ by side ind. hyp. ii. Since B is a *strict* part of A, we can apply the main ind. hyp. i and obtain $\Gamma \vdash r'\,u' \Uparrow C$.

```
  subst_x : {A:ty} wn S A -> {C} ({x:tm} {v:var x} wne x A v
                        -> wne (T x) C v) -> wn (T S) C -> type.

  subst_x_x   : subst_x A D A ([x][v][dx]dx) D.
  subst_x_app : subst_x A D C ([x][v][dx]
                        wne_app (E x v dx)
                               (F x v dx : wn (S x) B)) EF
                <- subst_x A D (B => C) E E'
                <- subst_wn A D F F'
                <- app_wn B F' E' EF.

  %reduces C <= A (subst_x A D C E F).
```

The %reduces declaration states that the type expression C is a subexpression of A. Twelf checks that this invariant is preserved in all possibilities of introducing subst_x A D C E F. In case subst_x_x it holds because C is instantiated to A. In case subst_x_app it follows from the ind. hyp. which states that already B => C is a subexpression of A.

(iv) Show $\Gamma \vdash [s/x]t \Downarrow^y C$ for $\Gamma, x : A \vdash t \Downarrow^y C$. There a two cases. $t = y$, which holds immediately, and $t = r\,u$, which follows from side ind. hyp.s ii and iv. In our Twelf representation, we cannot distinguish variable y from any other term, so we widen the first case to cover all t such that x is not free in t. This

is expressed by letting E not refer to x or dx.

```
subst_y : {A:ty} wn S A -> ({x:tm} {v:var x} wne x A v
        -> wne (T x) C Y) -> wne (T S) C Y -> type.

subst_y_y   : subst_y A D ([x][v][dx] E) E.
subst_y_app : subst_y A D ([x][v][dx]
          wne_app (E x v dx) (F x v dx)) (wne_app E' F')
              <- subst_y A D E E'
              <- subst_wn A D F F'.
```

$\square$

To justify the appeals to the ind. hyp.s we invoke the Twelf termination checker with the following termination order.

```
%terminates {(Ax Ay As Aa) (Ex Ey Es Ea)}
  (subst_x  Ax _ _ Ex _)
  (subst_y  Ay _   Ey _)
  (subst_wn As _   Es _)
  (app_wn   Aa _   Ea _).
```

It expresses that the four type families are mutually recursive and terminate w. r. t. the lexicographic order on pairs $(A, \mathcal{E})$ of types A and typing derivations $\mathcal{E}$. This corresponds on a main induction on A and a side induction on $\mathcal{E}$. To verify termination, Twelf makes use of the %reduces declaration. Using the world block wne_var, the code also coverage-checks, hence, the lemma is completely machine-verified.

Lemma 3.2 *If* $\mathcal{D} :: \Gamma \vdash t : A$ *then* $\Gamma \vdash t \Uparrow A$.

Proof. By induction on $\mathcal{D}$, using the application lemma. Can be directly formalized in Twelf [1]. $\square$

3.3 Soundness of Inductive Characterization

To complete our proof of weak normalization, we need to show that for each term t in the relation $t \Uparrow A$ or $t \Downarrow^x A$ there exists a normal form v such that $t \longrightarrow^*$ v. After formulating full β-reduction $\longrightarrow$ with the usual closure properties, the proof is a simple induction on the derivation $\mathcal{E} :: t \Uparrow A$ resp. $\mathcal{E} :: t \Downarrow^x A$. The formalization of multi-step β-reduction $\longrightarrow^*$ and fundamental properties like closure under substitution is completely standard and available online [1]. Similar proofs can be found in Schürmann's thesis [17, Ch. 3].

Typed β-normal forms can be defined inductively by two judgements $\Gamma \vdash t \uparrow A$, *t of type A is normal*, and $\Gamma \vdash t \downarrow A$, *t of type A is normal and neutral*, the last adjective meaning that $t = x\,\boldsymbol{s}$ for some $\boldsymbol{s}$.

$$\frac{(x\!:\!A) \in \Gamma}{\Gamma \vdash x \downarrow A} \text{ ne\_var} \qquad \frac{\Gamma \vdash r \downarrow A \to B \qquad \Gamma \vdash s \uparrow A}{\Gamma \vdash r\,s \downarrow B} \text{ ne\_app}$$

$$\frac{\Gamma \vdash r \downarrow A}{\Gamma \vdash r \uparrow A} \text{ nf\_ne} \qquad \frac{\Gamma, x\!:\!A \vdash t \uparrow B}{\Gamma \vdash \lambda xt \uparrow A \to B} \text{ nf\_lam}$$

Lemma 3.3 (Soundness)

(i) *If $\mathcal{D} :: \Gamma \vdash t \Downarrow^x A$ then $\Gamma \vdash t' \downarrow A$ for some $t' \overset{*}{\longleftarrow} t$.*

(ii) *If $\mathcal{D} :: \Gamma \vdash t \Uparrow A$ then $\Gamma \vdash t' \uparrow A$ for some $t' \overset{*}{\longleftarrow} t$.*

Proof. Simultaneously by induction on $\mathcal{D}$. The Twelf formalization is unproblematic. □

Theorem 3.4 (Weak normalization) *If $\Gamma \vdash t : A$ then $\Gamma \vdash t' \uparrow A$ for some $t' \overset{*}{\longleftarrow} t$.*

The theorem is the just the composition of lemmata 3.2 and 3.3.

Remark 3.5 η-long β-normal forms can be enforced by requiring A to be a base type in rule `nf_ne`. The expansion of a β-normal term t to an η-long form can be defined by induction on $\Gamma \vdash t \uparrow A$. Thus, our judgement $t \Uparrow A$ captures trivially also the weakly $\beta\eta$-normalizable terms.

4   On Proof-Theoretical Limitations of Twelf

Having successfully completed the proof of weak normalization we are interested whether it could be extended to strong normalization and stronger object theories, like Gödel's T. In this section, we touch these questions, but our answers are speculative and preliminary.

Joachimski and Matthes [9] extend their proof to Gödel's T, using the infinitary ω-rule to state when a recursive function over natural numbers is weakly normalizing. Their proof is not directly transferable since only finitary rules can be represented in Twelf.

For the same reason, Tait's proof cannot be formalized directly in Twelf. Its key part is the definition

$$\frac{\forall s.\ \ s \Uparrow A \ \Rightarrow \ r\,s \Uparrow B}{r \Uparrow A \to B}$$

with an infinitary premise. Its literal translation into Twelf

```
wn_arr : ({S:tm} wn S A -> wn (app R S) B) -> wn R (A => B)
```

means something else, namely "if for a fresh term `S` for which we assume `wn S A` it holds that `wn (app R S) B`, then `wn R (A => B)`". Translating this back into mathematical language, we obtain the rule

$$\frac{x \Uparrow A \ \Rightarrow \ r\,x \Uparrow B}{r \Uparrow A \to B} \quad \text{for a fresh variable } x.$$

Since variables x are weakly normalizing anyway, we can simplify the premise further to $r\,x \Uparrow B$, obtaining clearly something we did not want in the first place.

Recently, Sarnat and Schürmann [16] have proven weak normalization for the simply-typed λ-calculus in Twelf using logical relations. They first give a inductive characterization of weakly normalizing terms similar to ours. Instead of directly proving that it is closed under application (our Lemma 3.1) they follow a Tait-style logical relation argument. To make it work, they define minimal first-order

logic (MFOL) and prove cut elimination. The logical relation is then represented within MFOL, but they can show an escape lemma: each term which is provably in the logical relation is actually weakly normalizing. The proof complexity has been shifted to cut elimination; it is established by a lexicographic induction just as our application/substitution lemma.

Sarnat and Schürmann have formalized the first logical relations proof in Twelf, an impressive achievement, since it was unknown for a long time how to do this. Yet, we have given a much simpler proof of the same result. It is unclear whether their approach scales to stronger systems, e.g., Gödel's T. At least, the termination checker of Twelf must be extended to lift the proof-theoretic strength of Twelf beyond primitive recursive arithmetic.

5 Strong Normalization

Due to the interpretation of quantification in Twelf, infinitary rules cannot be represented, which also obstructs the definition of the predicate *strongly normalizing* sn by the inductive rule

$$\frac{\forall t'.\ t \longrightarrow t' \Rightarrow \mathsf{sn}\ t'}{\mathsf{sn}\ t},$$

expressing that the set of strongly normalizing terms is the accessible part of the reduction relation.

However, the inductive characterization of weakly normalizing terms can be modified to account for strongly normalizing terms. A term $(\lambda x t)\ s$ is weakly normalizing iff $[s/x]t$ is—this holds for strong normalization if we additionally require that s is strongly normalizing. Thus, in each weak head expansion step, the term s must be strongly normalizing. We inductively define such a expansion relation $\longrightarrow_{\mathsf{sn}}$ together with the set of strongly normalizing terms.

$$\frac{(x\,{:}\,A) \in \Gamma}{\Gamma \vdash x \Downarrow^x A}\ \mathtt{sne\_var} \qquad \frac{\Gamma \vdash r \Downarrow^x A \to B \qquad \Gamma \vdash s \Uparrow A}{\Gamma \vdash r\,s \Downarrow^x B}\ \mathtt{sne\_app}$$

$$\frac{\Gamma \vdash r \Downarrow^x A}{\Gamma \vdash r \Uparrow A}\ \mathtt{sn\_ne} \qquad \frac{\Gamma, x\,{:}\,A \vdash t \Uparrow B}{\Gamma \vdash \lambda x.t \Uparrow A \to B}\ \mathtt{sn\_lam}$$

$$\frac{\Gamma \vdash r \longrightarrow_{\mathsf{sn}} r' \qquad \Gamma \vdash r' \Uparrow A}{\Gamma \vdash r \Uparrow A}\ \mathtt{sn\_exp}$$

$$\frac{\Gamma \vdash s \Uparrow A}{\Gamma \vdash (\lambda x t)\,s \longrightarrow_{\mathsf{sn}} [s/x]t}\ \mathtt{sr\_beta} \qquad \frac{\Gamma \vdash r \longrightarrow_{\mathsf{sn}} r'}{\Gamma \vdash r\,s \longrightarrow_{\mathsf{sn}} r'\,s}\ \mathtt{sr\_app}$$

The soundness of the inductive characterization must be established on paper, but an application/substitution lemma can now be proved in Twelf analogously to Lemma 3.1, from which it follows that each well-typed term is strongly normalizing.

6   Conclusions

We have presented a formalization of Joachimski and Matthes' version of an elementary proof of weak normalization of the simply-typed λ-calculus in Twelf. We further have outlined a partial formalization of strong normalization proof and discussed feasibility of Tait-style proofs in Twelf.

Related work: Hereditary substitutions.

Watkins *et al.* [21] have formulated a concurrent logical framework where all terms are in normal form. Consequently, substitution and application must perform some evaluation. Watkin's substitution and application algorithms, called hereditary substitutions, match our Lemma 3.1 precisely. Recently, Lee, Crary, and Harper [10] have formalized a full intermediate language for SML based on hereditary substitutions.

Acknowledgments.

The author thanks Ralph Matthes, Frank Pfenning, Brigitte Pientka, Jason Reed, Carsten Schürmann and Kevin Watkins for discussions on the topic in the years 2000–2004. He is indebted to Thierry Coquand for comments on a draft, to Andrej Filinski and Carsten Schürmann to comments on the workshop version of this article, and to Chung-Chieh Shan for improvements of the Twelf code.

References

[1] Abel, A., *A Twelf proof of weak normalization for the simply-typed λ-calculus*, Twelf code, available on the author's homepage (2007).

[2] Altenkirch, T., *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, TLCA'93*, Lecture Notes in Computer Science **664** (1993), pp. 13–28.

[3] Berghofer, S., "Proofs, Programs and Executable Specifications in Higher-Order Logic," Ph.D. thesis, Technische Universität München (2003).

[4] Girard, J.-Y., Y. Lafont and P. Taylor, "Proofs and Types," Cambridge Tracts in Theoretical Computer Science **7**, Cambridge University Press, 1989.

[5] Goguen, H., *Typed operational semantics*, in: M. Deziani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculi and Applications (TLCA 1995)*, Lecture Notes in Computer Science **902** (1995), pp. 186–200.

[6] Harper, R., F. Honsell and G. Plotkin, *A Framework for Defining Logics*, Journal of the Association of Computing Machinery **40** (1993), pp. 143–184.

[7] Harper, R. and F. Pfenning, *On equivalence and canonical forms in the LF type theory*, ACM Transactions on Computational Logic **6** (2005), pp. 61–101.

[8] INRIA, "The Coq Proof Assistant Reference Manual," Version 8.0 edition (2004), http://coq.inria.fr/doc/main.html.

[9] Joachimski, F. and R. Matthes, *Short proofs of normalization*, Archive of Mathematical Logic **42** (2003), pp. 59–87.

[10] Lee, D. K., K. Crary and R. Harper, *Towards a mechanized metatheory of Standard ML*, in: M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007* (2007), pp. 173–184.

[11] Pfenning, F., *Logical frameworks*, in: J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, volume 2*, Elsevier and MIT Press, 2001 pp. 1063–1147.

[12] Pfenning, F. and C. Schürmann, *System description: Twelf - a meta-logical framework for deductive systems*, in: H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence **1632** (1999), pp. 202–206.

[13] Pientka, B., *Termination and reduction checking for higher-order logic programs*, in: R. Goré, A. Leitsch and T. Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001*, Lecture Notes in Artificial Intelligence **2083** (2001), pp. 401–415.

[14] Pollack, R., "The Theory of LEGO," Ph.D. thesis, University of Edinburgh (1994).

[15] Rohwedder, E. and F. Pfenning, *Mode and termination checking for higher-order logic programs*, in: H. R. Nielson, editor, *Programming Languages and Systems - ESOP'96, 6th European Symposium on Programming, Linköping, Sweden, April 22-24, 1996, Proceedings*, Lecture Notes in Computer Science **1058** (1996), pp. 296–310.

[16] Sarnat, J. and C. Schürmann, *A proof theoretic account of logical relations* (2005), submitted.

[17] Schürmann, C., "Automating the Meta-Theory of Deductive Systems," Ph.D. thesis, Carnegie-Mellon University (2000).

[18] Schürmann, C. and F. Pfenning, *A coverage checking algorithm for LF*, in: D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, Lecture Notes in Computer Science **2758** (2003), pp. 120–135.

[19] Valentini, S., *A note on a straightforward proof of normal form theorem for simply typed lambda-calculi*, Bollettino dell'Unione Matematica Italiana **8** (1994), pp. 207–213.

[20] van Raamsdonk, F., P. Severi, M. H. Sørensen and H. Xi, *Perpetual reductions in lambda calculus*, Information and Computation **149** (1999), pp. 173–225.

[21] Watkins, K., I. Cervesato, F. Pfenning and D. Walker, *A concurrent logical framework: The propositional fragment*, in: S. Berardi, M. Coppo and F. Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, Lecture Notes in Computer Science (2004), pp. 355–377.