A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F

Kevin Donnelly [1,2] and Hongwei Xi [1,3]

Computer Science Department, Boston University
Boston, USA

Abstract

We formalize in the logical framework ATS/LF a proof based on Tait's method that establishes the simply-typed lambda-calculus being strongly normalizing. In this formalization, we employ higher-order abstract syntax to encode lambda-terms and an inductive datatype to encode the reducibility predicate in Tait's method. The resulting proof is particularly simple and clean when compared to previously formalized ones. Also, we mention briefly how a proof based on Girard's method can be formalized in a similar fashion that establishes System F being strongly normalizing.

Key words: Logical frameworks, Normalization, Tait's method, Logical relations, Reducibility candidates, HOAS, ATS/LF

1 Introduction

ATS/LF [4] is a logical framework rooted in the Applied Type System [15] and is a pure total fragment of the programming language ATS. It uses a restricted form of dependent types in which types may only be indexed by terms drawn from limited domains in which equality is decidable (and can also be effectively reasoned about). ATS/LF supports the use of higher-order abstract syntax (HOAS) [9] to encode object languages. The use of HOAS, in which object variables are identified with metavariables and β-reduction models substitution, leads to particularly simple and elegant encodings. The combination of a limited type-index language and a powerful proof language, as found in ATS/LF, allows for inductive proofs of metatheorems over full higher-order abstract syntax to be directly encoded as total recursive functions. The use of inductive datatypes with negative occurrences allows for the encoding of the reducibility predicate.

[1] The work is partly funded by NSF grant CCR-0229480
[2] Email: kevind@cs.bu.edu
[3] Email: hwxi@cs.bu.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

In this paper, we formalize a proof of strong normalization of the simply typed lambda-calculus (STLC) using Tait's method, closely following the one in [7]. On one hand, we use HOAS to encode lambda-terms, obviating the need for explicitly manipulating substitution on such terms. On the other hand, we use first-order abstract syntax (FOAS) to encode typing derivations in STLC, which conveniently supports inductive reasoning on typing derivations.

To our knowledge this is the first formalized (or mechanized) proof of strong normalization using Tait's method for an object language defined with HOAS. When compared to other formalized proofs of strong normalization in the literature, the brevity of our formalized proof and its closeness to the concise and elegant proof in [7] yield some concrete evidence in support of the effectiveness of the representation of STLC in ATS/LF. To further strengthen this claim, we also discuss the extension to the case of System F, formalizing a proof of strong normalization of System F based on Girard's notion of reducibility candidates [6]. We expect that the techniques developed here can also allow for the formalization of other proofs by logical relations while still being able to take advantage of HOAS.

2 ATS/LF

ATS/LF is split into two main parts: the language of types and type indices (called the *statics*), and the language of proofs (called the *dynamics*). The statics is basically simply-typed lambda-calculus with constants (but no recursion), and terms in the statics are referred to as *static terms* and types in the statics are referred to as *sorts*. There are three important built-in base sorts:

- *prop* : A sort for static terms which represent types of proofs.
- *int* : A sort for static integer terms. There are constants for each integer $(\ldots, \mathit{-1}, \mathit{0}, \mathit{1}, \ldots : \mathit{int})$ and for addition $(+ : (\mathit{int}, \mathit{int}) \rightarrow \mathit{int})$ and subtraction $(- : (\mathit{int}, \mathit{int}) \rightarrow \mathit{int})$.
- *bool* : A sort for static boolean conditions. There are constants for truth values $(\mathit{true}, \mathit{false} : \mathit{bool})$ and equality and inequality on integers $(=, < : (\mathit{int}, \mathit{int}) \rightarrow \mathit{bool})$.

Static constants may take multiple arguments. Equality in the statics is basically β-conversion plus Presburger arithmetic, and it is decided by converting to $\beta\eta$ long normal form and then using a decision procedure for integer (in)equalities (after mapping boolean terms to integer terms).

The dynamics is a dependently typed language with well-founded recursion, exhaustive case-analysis and inductive datatypes. Termination is checked using a programmer-supplied metric, which is a tuple of static terms representing natural numbers and decreasing in each recursive call according to the standard lexicographic ordering. Please see [13] for more details on this style of termination checking. Case coverage is checked by requiring that any unlisted cases introduce assumptions that allow *false* to be proven [14]. In the concrete syntax, a proof (function) declaration looks like:

Syntax:

$$\begin{array}{lll} \text{terms} & t \in tm ::= x \mid \lambda x.t \mid t_1\ t_2 \mid \mathsf{c} \\ \text{types} & \tau \in tp ::= \mathsf{B} \mid \tau_1 \to \tau_2 \\ \text{contexts} & \Gamma \in ctx ::= \cdot \mid \Gamma, x : \tau \end{array}$$

Fig. 1. Syntax for Simply-typed λ-calculus

```
prfun proofName {x1:stx1, ..., xn:stxn} .<m1, ..., mk>.
      (p1:T1, ..., pl:Tl) : [y1:sty1, ..., ym:stym] T = ...
```

This declaration is for a total recursive function called *proofName* (`prfun` is a keyword for introducing proof functions) with the type:

$$\forall x_1 : stx_1, \ldots, \forall x_n : stx_n.(T_1, \ldots, T_l) \to \exists y_1 : sty_1, \ldots, \exists y_m : sty_m.T$$

This type signature consists of four parts. First, there are n static parameters x_i of sorts stx_i, enclosed in curly braces (think of these as universally quantified). Second, there is a metric, enclosed in `.<` and `>.`, which is a k-tuple of static terms representing natural numbers and may contain $x_1, \ldots, x_n$. Third, there are l dynamic parameters p_i with types T_i that may contain $x_1, \ldots, x_n$. Fourth, there is the return type which consists of m existentially quantified static variables y_i of sorts sty_i and a type T which may contain $x_1, \ldots, x_n, y_1, \ldots, y_m$. In the case where the declared function *proofName* is not recursive, we may also use the keyword `prfn` and give no metric. Please see [4,5] for some examples of proofs formed in ATS/LF.

3 Encoding the Object Language

3.1 Syntax

The object language for which we prove strong normalization is STLC with a constant c and a base type B. The syntax of the language is shown in Figure 1. We will encode the syntax in the statics using HOAS. In order to do so we declare a static sort for each syntactic category. We begin with a sort, tm, with constructors for each term constructor of the object language:

$$TMlam : (tm \to tm) \to tm \qquad TMapp : (tm, tm) \to tm \qquad TMcst : tm$$

Object variables are encoded as metavariables. The constant *TMcst* is only used in the formalization as a placeholder when recursing under lambda binders. Object functions are represented by functions in the statics, and this allows us to model substitution in the object language with application in the meta-language. The terms of the object language are encoded in the statics with the function $\ulcorner \cdot \urcorner$ defined by:

$$\begin{array}{ll} \ulcorner x \urcorner = x & \ulcorner \mathsf{c} \urcorner = TMcst \\ \ulcorner \lambda x.t \urcorner = TMlam(\lambda x.\ulcorner t \urcorner) & \ulcorner t_1\ t_2 \urcorner = TMapp(\ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner) \end{array}$$

This is a compositional bijection between terms of the object language with up to n free variables and static terms of sort tm with up to n free variables.

Reduction: $t_1 \longrightarrow t_2$

$$\frac{t \longrightarrow t'}{\lambda x.t \longrightarrow \lambda x.t'}\ (\mathit{REDlam}) \qquad \frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ (\mathit{REDapp1})$$

$$\frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}\ (\mathit{REDapp2}) \qquad \frac{}{(\lambda x.t_1)\ t_2 \longrightarrow t_1[t_2/x]}\ (\mathit{REDapp3})$$

Fig. 2. Reduction rules for λ-calculus

To encode types we declare a sort tp, with constructors for each type constructor of the object language:

$$\mathit{TPbas} : \mathit{tp} \qquad \mathit{TPfun} : (\mathit{tp}, \mathit{tp}) \rightarrow \mathit{tp}$$

In some encodings with HOAS, there is no explicit representation of contexts in the representation of typing judgments, but instead the context of the metalanguage is utilized. Such higher-order representations of the typing judgment, as often used in Twelf [10], benefit from inheriting substitution on typings from the metalanguage, and so do not need a typing substitution lemma. On the other hand, the use of explicit contexts allows for a first-order representation of typing derivations. This, along with the separation between statics and dynamics, allows us to prove metatheorems directly, using total recursive functions, while still taking advantage of HOAS for object syntax. The inconvenience of having to prove substitution on typing derivations is minor, and not pervasive as issues involving binders in the syntax are. In fact, we do not ever need to make use of substitution on typing derivations in the proof of strong normalization. Contexts, of sort ctx, are represented by lists of pairs of a tm and a tp:

$$\mathit{CTXnil} : \mathit{ctx} \qquad \mathit{CTXcons} : (\mathit{tm}, \mathit{tp}, \mathit{ctx}) \rightarrow \mathit{ctx}$$

We may sometimes abbreviate *CTXcons(t,T,G)* as $(t, T) :: G$. Really this sort represents explicitly typed substitutions. A term of sort ctx only represents a well-formed context if its tm subterms are all distinct metavariables. We will return to this issue when we encode typing derivations.

3.2 Reduction

The rules for small-step reduction for pure λ-calculus are shown in Figure 2. Reduction, $t \longrightarrow t'$, is encoded as a datatype with type constructor $\mathit{RED} : (\mathit{tm}, \mathit{tm}, \mathit{int}) \rightarrow \mathit{prop}$ (where the third index measures the size of the derivation) and one term constructor to encode each rule in Figure 2. The most interesting rules are *REDlam* and *REDapp3* which correspond to the dynamic term constructors:

$$\begin{aligned}\mathit{REDlam} &: \forall f : \mathit{tm} \rightarrow \mathit{tm}.\forall f' : \mathit{tm} \rightarrow \mathit{tm}.\forall n : \mathit{nat}.\\ &\qquad (\forall x : \mathit{tm}.\ \mathit{RED}(f\ x, f'\ x, n)) \rightarrow \mathit{RED}(\mathit{TMlam}\ f, \mathit{TMlam}\ f', n+1)\\ \mathit{REDapp3} &: \forall f : \mathit{tm} \rightarrow \mathit{tm}.\forall t : \mathit{tm}.\ \mathit{RED}(\mathit{TMapp}(\mathit{TMlam}\ f, t), f\ t, 0)\end{aligned}$$

Since the rules themselves are first order, adequacy follows from the fact that the higher-order syntax in the type indices correspond to the right terms. The most interesting rule is *REDlam*: from the quantification in the argument of the constructor $(\forall x : \mathit{tm}.\ \mathit{RED}(f\ x, f'\ x, n))$ and the fact that application

Type formation: $\vdash \tau\ type$

$$\frac{}{\vdash \mathsf{B}\ type}\ (TPbas) \qquad \frac{\vdash \tau_1\ type \quad \vdash \tau_2\ type}{\vdash \tau_1 \to \tau_2\ type}\ (TPfun)$$

Typing: $\Gamma \vdash t : \tau$

$$\frac{(x:\tau) \in \Gamma \quad \vdash \tau\ type}{\Gamma \vdash x : \tau}\ (DERvar)$$

$$\frac{\Gamma, x:\tau_1 \vdash t : \tau_2 \quad \vdash \tau_1\ type}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}\ (DERlam) \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ (DERapp)$$

Fig. 3. Typing rules for Simply-typed λ-calculus

in the statics models substitution, we can see that $f\ x$ and $f'\ x$ represent lambda-terms with x being free and that $TMlam\ f$ and $TMlam\ f'$ represent these same terms with x bound by a lambda.

3.3 Type Assignment

The rules for typing judgments are shown in Figure 3. We begin by defining the context lookup relation $(x : \tau) \in \Gamma$. For this we use a datatype with type constructor $INCTX : (tm, tp, ctx, int) \to prop$, where $INCTX(t, T, G, n)$ means that (t, T) is at the n^{th} index in G (abbreviated as $(t, T) \in_n G$), and two term constructors which correspond to the rules:

$$\frac{}{(t,T) \in_0 ((t,T) :: G)}\ (INCTXone) \qquad \frac{(t,T) \in_n G}{(t,T) \in_{n+1} ((t',T') :: G)}\ (INCTXshi)$$

Note that if $INCTX(t, T, G, n)$ is inhabited, its member is unique and isomorphic to n (since it is a non-branching tree of depth n).

We encode the judgment $\vdash \tau\ type$ with a datatype, where the type constructor is $TP : (tp, int) \to prop$ and the term constructors represent the following rules (where we write $\vdash_n T\ type$ for $TP(T, n)$):

$$\frac{}{\vdash_0 TPbas\ type}\ (TPbas) \qquad \frac{\vdash_{n_1} T_1\ type \quad \vdash_{n_2} T_2\ type}{\vdash_{n_1+n_2+1} TPfun(T_1, T_2)\ type}\ (TPfun)$$

While the constructors of this type have the same names as terms of sort tp, there is no ambiguity because dynamic terms are strictly separated from static terms. The type $TP(T, n)$ contains a single element which is isomorphic to T if the size of T is n. The size index is used to provide a metric to support induction on the structure of types. For convenience, we define $TP0(T) \equiv \exists n : nat.\ TP(T, n)$ (which we abbreviate as $\vdash T\ type$).

The encoding of the typing judgment $\Gamma \vdash t : \tau$ is a dependent datatype, $DER : (ctx, tm, tp, int) \to prop$, where the last index is a measure of the size of the typing derivation. The constructors correspond to the inference rules in Figure 4 (where $G \vdash_n t : T$ abbreviates $DER(G, t, T, n)$). The typing rule for variables is encoded by the term constructor:

$$DERvar : \forall G : ctx. \forall t : tm. \forall T : tp. \forall n : nat.\ (INCTX(t, T, G, n), TP0\ T) \to DER(G, t, T, 0)$$

The context is represented as a list, so the variable lookup identifies the index in the list that corresponds to the given variable. The typing rule for lambda-

Encoded Typing: $G \vdash_n t : T$

$$\frac{(t,T) \in_n G \quad \vdash T\ type}{G \vdash_0 t : T}\ (DERvar)$$

$$\frac{\vdash T_1\ type \quad (\forall x.\ (x,T_1) :: G \vdash_n f\ x : T_2)}{G \vdash_{n+1} TMlam\ f : TPfun(T_1,T_2)}\ (DERlam)$$

$$\frac{G \vdash_{n_1} t_1 : TPfun(T_1,T_2) \quad G \vdash_{n_2} t_2 : T_1}{G \vdash_{n_1+n_2+1} TMapp(t_1,t_2) : T_2}\ (DERapp)$$

Fig. 4. Encoded Typing Rules

abstraction is encoded by the following constructor:

$$\begin{aligned} DERlam : \forall G : ctx.\forall f : tm \to tm.\forall T_1 : tp.\forall T_2 : tp.\forall n : nat.\forall l : nat.\\ (TP0\ T_1, \forall x.\ DER(CTXcons(x,T_1,G), f\ x, T_2, n)) \to\\ DER(G, TMlam\ f, TPfun(T_1,T_2), n+1) \end{aligned}$$

Note that the quantification over x in the second argument of this constructor $(\forall x.DER(CTXcons(x,T_1,G), f\ x, T_2, n))$ guarantees that x is a metavariable not occurring in G and thus $CTXcons(x,T_1,G)$ is a well-formed context if G is. The typing rule for application is encoded by the following constructor:

$$\begin{aligned} DERapp : \forall G : ctx.\forall t_1 : tm.\forall t_2 : tm.\forall T_1 : tp.\forall T_2 : tp.\forall n_1 : nat.\forall n_2 : nat.\\ (DER(G,t_1,TPfun(T_1,T_2),n_1), DER(G,t_2,T_1,n_2)) \to\\ DER(G, TMapp(t_1,t_2), T_2, n_1+n_2+1) \end{aligned}$$

For convenience we also define $DER0(G,t,T) \equiv \exists n : nat.\ DER(G,t,T,n)$. This representation for typing derivations is quite interesting. The dynamic terms inhabiting the datatype $DER0(G,t,T)$ are isomorphic to simply-typed lambda-terms of Church-style in which variables are represented as de Bruijn indices. The context G is a typed substitution, which we can decompose into a substitution $\Theta = \langle t_1, \ldots, t_m \rangle$ (which maps the ith variable to t_i for $1 \leq i \leq m$) and a context $\Gamma = \langle T_1, \ldots, T_m \rangle$. The datatype $DER0(G,t,T)$ really represents a hypothetical judgment saying that if we have derivations of $\vdash t_i : T_i$ (for $1 \leq i \leq m$) then we can form a derivation of $\vdash t : T$. As long as Θ is a list of distinct meta-variables (say $\langle x_1, ..., x_m \rangle$), this is an adequate encoding of the usual typing judgment $x_1 : T_1, ..., x_m : T_m \vdash t : T$. We can guarantee that a context is well-formed in this way when it is empty or when it appears in a derivation that is a sub-derivation of one with an empty context. We are able to prove strong normalization for terms typed in the empty context and, since reduction under lambda is allowed, this implies strong normalization for terms containing free variables as well.

4 Strong Normalization Proof

In this section, we formalize a proof of strong normalization of STLC based on Tait's method [12]. The formalized proof is nearly identical to the one in [7], with the only exception that we use the constant `c` in some places where the proof in [7] uses a variable. The cause for this exception directly results

from HOAS being chosen for representing lambda-terms (and thus making it difficult to manipulate object variables). The proofs for the final few lemmas and strong normalization theorem are given in Appendix A and the entire proof can be found on-line:

`http://www.cs.bu.edu/~hwxi/ATS/EXAMPLE/LF/STLC-SN-hoas.dats`

Definition 4.1 [Strong Normalization] A term t is strongly normalizing with bound n, written $\mathsf{SN}_n(t)$, if for all t' such that $t \longrightarrow t'$ we have $\mathsf{SN}_{n'}(t')$ for some natural number $n' < n$ (i.e. all reduction sequences starting from t have length at most n). A term t is strongly normalizing, written $\mathsf{SN0}(t)$, if there is some n such that $\mathsf{SN}_n(t)$.

$\mathsf{SN}_n(t)$ is encoded using a dependent datatype with type constructor $SN : (tm, int) \rightarrow prop$ and one term constructor of the same name:

$$SN : \forall t : tm.\forall n : nat.(\forall t' : tm.RED0(t, t') \rightarrow \exists n' < n.\ SN(t', n')) \rightarrow SN(t, n)$$

We encode $\mathsf{SN0}(t)$ by defining $SN0(t) \equiv \exists n : nat.\ SN(t, n)$. Strong normalization is closed under forward and backward reduction.

Lemma 4.2 *If* $\mathsf{SN}_n(t)$ *and* $t \longrightarrow t'$ *then* $\mathsf{SN}_{n'}(t')$ *for some* $n' < n$.

Proof. *This follows directly from the definition of* $\mathsf{SN}_n(t)$. □

The ATS/LF proof for this lemma is given as follows:

```
prfn forwardSN {t:tm, t':tm, n:nat}
  (sn: SN(t, n), red: RED0(t, t')) : [n':nat | n' < n] SN(t', n') =
  let prval SN (fsn) = sn in fsn red end
```

The keyword `prval` here is similar to the keyword `val` in ML.

Lemma 4.3 *If for all* t', $t \longrightarrow t'$ *implies* $\mathsf{SN0}(t')$, *then* $\mathsf{SN0}(t)$.

Proof. *For any* t *there are a finite number of* t' *such that* $t \longrightarrow t'$. *For each of these* t' *we have* $\mathsf{SN}_{n'}(t')$ *for some* n'. *If we take* n *to be one plus the maximum of these* n' *(which exists because there are only finitely many) then we have* $\mathsf{SN}_n(t)$ *so* $\mathsf{SN0}(t)$. □

This is an obvious consequence of the definition of *SN0* and the fact that each term has a finite number of different reducts, and formalizing it in ATS/LF is entirely uninspiring (as the argument is purely set-theoretic). So we use the keyword `dynprf` to introduce it as an unproven lemma:

```
dynprf backwardSN : {t:tm} ({t':tm} RED0 (t, t') -> SN0 t') -> SN0 t
```

This is the only unproven lemma in the entire formalization.

Attempting to directly prove strong normalization of well-typed terms by induction on typing derivations does not work because the induction hypothesis is not strong enough to handle application terms. In order to make the proof go through, we strengthen the induction hypothesis using the notion of *reducibility*, introduced by Tait [12].

Definition 4.4 [Reducibility] A lambda-term t is reducible at a type τ, written $\mathsf{R}_\tau(t)$, if:

(i) τ is a base type (that is, B in our case) and $\mathsf{SN0}(t)$, or

(ii) τ is $\tau_1 \to \tau_2$ and for all t', $\mathsf{R}_{\tau_1}(t')$ implies $\mathsf{R}_{\tau_2}(t\ t')$.

It should be emphasized that $\mathsf{R}_\tau(t)$ does not necessarily imply that t can be assigned the type τ. As a matter of fact, we have $\mathsf{R}_\mathsf{B}(\omega)$ for $\omega = \lambda x.xx$ according to the definition. Also, it is clear that we cannot have $\mathsf{R}_{\mathsf{B}\to\mathsf{B}}(\omega)$ as it would otherwise imply $\mathsf{R}_\mathsf{B}(\omega\omega)$, which is a contradiction since $\omega\omega$ is not normalizing.

The definition in ATS/LF uses a dependent datatype with type constructor $R : (tm, tp) \to prop$ and two term constructors:

$$\begin{aligned} &Rbas : \forall t : tm.\ SN0\ t \to R(t, TPbas) \\ &Rfun : \forall t : tm.\forall T_1 : tp.\forall T_2 : tp. \\ &\qquad (\forall t_1 : tm.R(t_1, T_1) \to R(TMapp(t, t_1), T_2)) \to R(t, TPfun(T_1, T_2)) \end{aligned}$$

This is not a positive datatype because there is a negative occurrence of R in the function case. However, this definition is still well-founded because the tp index is structurally decreasing in all recursive occurrences (both positive and negative). This allows us to view the datatype as being built up inductively in levels stratified by the tp index. In particular, this means that when we are building the level corresponding to $TPfun(T_1, T_2)$, the levels corresponding to T_1 and T_2 are already complete and thus the set of functions from level T_1 to level T_2 (which are the possible arguments of $Rfun$) is also complete.

We begin by proving some important properties of the reducibility predicate. We first define neutral terms as follows.

Definition 4.5 [Neutrality] A term is neutral if it is either the constant c or an application of the form $t\ t'$.

This is defined in ATS/LF as a dependent datatype with type constructor $NEU : tm \to prop$ and term constructors:

$$NEUcst : NEU(TMcst) \qquad NEUapp : \forall t : tm.\forall t' : tm.\ NEU(TMapp(t, t'))$$

We can now state and prove four important properties of reducibility, which are given the names CR 1-4 in [7]:

CR 1: If $\mathsf{R}_\tau(t)$ then $\mathsf{SN0}(t)$,

CR 2: If $\mathsf{R}_\tau(t)$ and $t \longrightarrow t'$ then $\mathsf{R}_\tau(t')$,

CR 3: If t is neutral and for all t', $t \longrightarrow t'$ implies $\mathsf{R}_\tau(t')$, then $\mathsf{R}_\tau(t)$, and

CR 4: $\mathsf{R}_\tau(\mathsf{c})$ for any τ, which is a special case of CR 3.

We first prove CR 2 on its own, and then prove CR 1, 3 and 4 simultaneously.

Lemma 4.6 (CR 2) ***Proof.*** *By induction on τ:*

case: *$\tau = \mathsf{B}$, so we have $\mathsf{SN0}(t)$. By closure of strong normalization under forward reduction (Lemma 4.2) we have $\mathsf{SN0}(t')$, so $\mathsf{R}_\mathsf{B}(t')$.*

case: *$\tau = \tau_1 \to \tau_2$, so for all t_1, $\mathsf{R}_{\tau_1}(t_1)$ implies $\mathsf{R}_{\tau_2}(t\ t_1)$. Fix any t_1 such that $\mathsf{R}_{\tau_1}(t_1)$, then we have $\mathsf{R}_{\tau_2}(t\ t_1)$ and since $t\ t_1 \longrightarrow t'\ t_1$, by induction hypothesis, we have $\mathsf{R}_{\tau_2}(t'\ t_1)$. Therefore $\mathsf{R}_{\tau_1\to\tau_2}(t')$.*

□

The proof is encoded in ATS/LF as follows:

```
prfun cr2 {t:tm, t':tm, T:tp, n:nat} .<n>.
  (tp: TP (T, n), r: R(t, T), rd : RED0(t, t')): R(t', T) =
  case* r of // [case*] indicates exhaustive pattern matching
    | Rbas (sn) => Rbas (forwardSN (sn, rd))
    | Rfun{_, T1, _} (fr) => let
        prval TPfun (_, tp2) = tp
      in
        Rfun(lam {t1:tm} (r:R(t1,T1)) => cr2(tp2, fr r, REDapp1 rd))
      end
```

This proof function is a fairly straightforward encoding of the argument, taking the extra argument of type $TP(T, n)$ to provide a termination metric. The proof has a slightly unusual feature: the *Rfun* case binds the static argument T_1 in order to be able to provide the type for the lambda-bound variable r.

Lemma 4.7 (CR 1, 3, 4) ***Proof.*** *We prove CR 1, CR 3, CR 4, in that order, by induction on τ. The argument for CR 3 makes use of a nested induction, and CR 4 follows directly from CR 3 at each level.*

case: $\tau = \mathsf{B}$. *Reducibility at base types is just strong normalization.*
CR 1: *Direct from the definition of* $\mathsf{R}_{\mathsf{B}}(\cdot)$.
CR 3: *By Lemma 4.3.*

case: $\tau = \tau_1 \to \tau_2$.
CR 1: *Let t be a term with* $\mathsf{R}_{\tau_1 \to \tau_2}(t)$. *By CR 4 induction hypothesis,* $\mathsf{R}_{\tau_1}(\mathsf{c})$, *therefore* $\mathsf{R}_{\tau_2}(t\ \mathsf{c})$. *By CR 1 induction hypothesis $t\ \mathsf{c}$ is SN and any reduction of t induces a reduction of $t\ \mathsf{c}$, so t is SN.*
CR 3: *Let t be neutral such that for all t' with $t \longrightarrow t'$ we have* $\mathsf{R}_{\tau_1 \to \tau_2}(t')$. *Let t_1 be a term such that* $\mathsf{R}_{\tau_1}(t_1)$, *we need to show* $\mathsf{R}_{\tau_2}(t\ t_1)$. *By CR 1 induction hypothesis we know* $\mathsf{SN}_n(t_1)$ *for some n and we continue by nested induction on n. $t\ t_1$ is neutral, so if we show that all terms that it reduces to are reducible, then we can use CR 1 induction hypothesis to conclude* $\mathsf{R}_{\tau_2}(t\ t_1)$. *Suppose $t\ t_1 \longrightarrow t_2$:*
case: $t_2 = t'\ t_1$, *with $t \longrightarrow t'$. We know* $\mathsf{R}_{\tau_1 \to \tau_2}(t')$ *and* $\mathsf{R}_{\tau_1}(t_1)$, *so we have* $\mathsf{R}_{\tau_2}(t'\ t_1)$.
case: $t_2 = t\ t_1'$ *with* $t_1 \longrightarrow t_1'$. *By CR 2 induction hypothesis* $\mathsf{R}_{\tau_1}(t_1')$, *and by Lemma 4.2,* $\mathsf{SN}_{n'}(t_1')$ *for some* $n' < n$, *so by induction* $\mathsf{R}_{\tau_2}(t\ t_1')$.
These are the only possibilities because t is neutral.

□

The full ATS/LF proof of this is omitted for brevity; it consists of 4 mutually recursive proof functions:

$$cr1 : \forall t : tm. \forall T : tp. \forall n : \mathrm{nat}.\ (TP(T, n), R(t, T)) \to SN0(t)$$

$$cr3 : \forall t : tm. \forall T : tp. \forall n : \mathrm{nat}.\ (NEU(t), TP(T, n), \forall t'.\ RED0(t, t') \to R(t', T)) \to R(t, T)$$

$$\begin{aligned} cr3a : \forall t : tm. \forall t_1 : tm. \forall T_1 : tp. \forall T_2 : tp. \forall m : \mathrm{nat}. \forall n_1 : \mathrm{nat}. \forall n_2 : \mathrm{nat}.\\ (TP(T_1, n_1), TP(T_2, n_2), NEU(t), R(t_1, T_1), SN(t_1, m),\\ \forall t'.\ RED0(t, t') \to R(t', TPfun(T_1, T_2))) \to R(TMapp(t, t1), T2) \end{aligned}$$

$$cr4 : \forall T : tp. \forall n : \mathrm{nat}.\ TP(T, n) \to R(TMcst, T)$$

Each of these functions takes arguments of the form $TP(T, n)$ in order to provide a metric that corresponds to structural recursion on T. The auxiliary lemma *cr3a* performs the inner induction on the length of the strong normalization bound of t_1, which is provided by its argument of type $SN(t_1, m)$.

Lemma 4.8 *If for all reducible t at type τ_1, $\mathsf{R}_{\tau_2}(t_1[t/x])$, then $\mathsf{R}_{\tau_1\to\tau_2}(\lambda x.t_1)$.*

Proof. *Assume $\mathsf{R}_{\tau_1}(t)$. By CR 1, we know there is n_1 such that $\mathsf{SN}_{n_1}(t_1[\mathsf{c}/x])$ (and therefore $\mathsf{SN}_{n_1}(t_1)$) and n_2 such that $\mathsf{SN}_{n_2}(t)$. We now proceed by induction on $n_1 + n_2$ to prove that $\mathsf{R}_{\tau_2}((\lambda x.t_1)\ t)$. We will show that $(\lambda x.t_1)\ t \longrightarrow t'$ implies $\mathsf{R}_{\tau_2}(t')$ for every t'. There are three possibilities.*

- *$(\lambda x.t_1)\ t$ reduces to $t_1[t/x]$, which is reducible by the hypothesis of the lemma.*
- *$(\lambda x.t_1)\ t$ reduces to $(\lambda x.t_1)\ t'$ with $t \longrightarrow t'$. By CR 2, $\mathsf{R}_{\tau_1}(t')$ and by Lemma 4.2 there is $n' < n$ with $\mathsf{SN}_{n'}(t')$, and thus we have $\mathsf{R}_{\tau_2}((\lambda x.t_1)\ t')$ by induction.*
- *$(\lambda x.t_1)\ t$ reduces to $(\lambda x.t_1')\ t$ with $t_1 \longrightarrow t_1'$. By CR 2, $t_1'[t/x]$ is reducible for any reducible t and the strong normalization bound of $(\lambda x.t_1')$ is less than $(\lambda x.t_1)$. So $(\lambda x.t_1')\ t$ is reducible by induction.*

Note that $(\lambda x.t_1)\ t$ is neutral. By CR 3, we have $\mathsf{R}_{\tau_2}((\lambda x.t_1)\ t)$. Since $\mathsf{R}_{\tau_2}((\lambda x.t_1)\ t)$ holds for every t satisfying $\mathsf{R}_{\tau_1}(t)$, we have $\mathsf{R}_{\tau_1\to\tau_2}(\lambda x.t_1)$ by definition. □

The formalization of this proof in ATS/LF is a total recursive function with the type:

$$\begin{aligned} absSound : \forall f : tm \to tm.\forall T_1 : tp.\forall T_2 : tp.\\ (TP0(T_1), TP0(T_2), \forall t : tm.R(t, T_1) \to R(f\ t, T_2)) \to\\ R(TMlam\ f, TPfun(T_1, T_2)) \end{aligned}$$

The proof closely follows the informal one given above, taking additional arguments of types $TP0(T_1)$ and $TP0(T_2)$, which are needed in calls to *cr2* and *cr3*. It also makes a call to the proof function *reduceFun* to perform the inner induction on the sum of the normalization bounds ($n_1 + n_2$ in the informal proof).

Now we can prove the main reducibility lemma which states that, given a term t, with a typing $\Gamma \vdash t : T$ and a substitution Θ such that for $x \in \mathrm{dom}(\Gamma)$, $\Theta(x)$ is reducible at type $\Gamma(x)$, then $t[\Theta]$, the result of applying Θ to t, is reducible at type T.

Lemma 4.9 *Let t be a term with $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$. If $t_1, \ldots, t_n$ are terms such that $\mathsf{R}_{\tau_i}(t_i)$ (for $1 \le i \le n$) then $\mathsf{R}_{\tau}(t[t_1/x_1, \ldots, t_n/x_n])$.*

Proof. *By induction on the derivation of $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$. We write $t[\underline{t}/\underline{x}]$ for $t[t_1/x_1, \ldots, t_n/x_n]$.*

$t = x_i$: Then $t[\underline{t}/\underline{x}] = t_i$ and $\tau = \tau_i$ and by hypothesis $\mathsf{R}_{\tau_i}(t_i)$.

$t = t'\ t''$: Then, by induction hypothesis, $\mathsf{R}_{\tau'\to\tau}(t'[\underline{t}/\underline{x}])$ and $\mathsf{R}_{\tau'}(t''[\underline{t}/\underline{x}])$. By the definition of $\mathsf{R}_{\tau}((t'[\underline{t}/\underline{x}])\ (t''[\underline{t}/\underline{x}]))$ and $(t'[\underline{t}/\underline{x}])\ (t''[\underline{t}/\underline{x}]) = (t'\ t'')[\underline{t}/\underline{x}]$.

$t = \lambda x.t'$: (assume x is fresh with respect to $x_1, \ldots, x_n$ and $t_1, \ldots, t_n$) Then τ is of the form $\tau'' \to \tau'$. Fix t'' such that $\mathsf{R}_{\tau''}(t'')$. By induction hypothesis,

$\mathsf{R}_{\tau'}(t'[\underline{t}/\underline{x}, t''/x])$. *By Lemma 4.8,* $\mathsf{R}_{\tau''\to\tau'}(\lambda x.t'[\underline{t}/\underline{x}])$, *and by the freshness of* x, $(\lambda x.t'[\underline{t}/\underline{x}]) = (\lambda x.t')[\underline{t}/\underline{x}]$.

□

When we prove this lemma in ATS/LF, the higher-order encoding buys us quite a bit over a first-order encoding. Because of HOAS, we do not have to think about freshness of variables nor do we have to explicitly prove that the substitution commutes with the lambda binding when handling the lambda case. Lemma 4.9 is encoded in ATS/LF as a total function, which we omit for brevity:

$$\mathit{reduceLemma} : \forall G : \mathit{ctx}.\forall t : \mathit{tm}.\forall T : \mathit{tp}.\forall n : \mathit{nat}.\ (\mathit{DER}(G, t, T, n), \mathit{RS0}(G)) \to R(t, T)$$

Note that $\mathit{RS0}(G)$ is a datatype that associates with each (t_i, T_i) in G, a proof of the reducibility predicate $R(t_i, T_i)$. Also note that we take advantage of the representation of contexts as typed substitutions to state the lemma. It is now a simple matter to prove strong normalization for closed terms using Lemma 4.9 and CR 1.

$$\mathit{normalize} : \forall t : \mathit{tm}.\forall T : \mathit{tp}.\ \mathit{DER0}(\mathit{CTXnil}, t, T) \to \mathit{SN0}(t)$$

It is easy to see that this implies strong normalization for open terms as well, because any reduction on a term with free variables corresponds to a reduction in the closed term formed by abstracting these variables.

5 Strong Normalization for System F

We have also formalized a proof of strong normalization for (the Curry-style version of) System F, which can be found on-line:

`http://www.cs.bu.edu/~hwxi/ATS/EXAMPLE/LF/F-SN-hoas.dats`

The terms and reduction rules for the language are the same as for STLC. The types of System F are given by:

$$\tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau$$

The types are encoded with a first-order representation using de Bruijn indices:

$$\mathit{TPvar} : \mathit{int} \to \mathit{tp} \qquad \mathit{TPfun} : (\mathit{tp}, \mathit{tp}) \to \mathit{tp} \qquad \mathit{TPall} : \mathit{tp} \to \mathit{tp}$$

This representation means that we have to spend a great deal of effort proving lemmas about renumbering and substitution. However, we do not know if it is possible to prove strong normalization using a higher-order representation for types.

We extend the type well-formedness judgment $\vdash \tau\ \mathit{type}$ to include a context: $\Delta \vdash \tau\ \mathit{type}$, and list the new rules as follows:

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha\ \mathit{type}}\ (\mathit{TPvar}) \qquad \frac{\Delta \vdash \tau_1\ \mathit{type} \quad \Delta \vdash \tau_2\ \mathit{type}}{\Delta \vdash \tau_1 \to \tau_2\ \mathit{type}}\ (\mathit{TPfun}) \qquad \frac{\Delta, \alpha \vdash \tau\ \mathit{type}}{\Delta \vdash \forall\alpha.\ \tau\ \mathit{type}}\ (\mathit{TPall})$$

Typing judgments are extended to include the extra context and there are also two additional typing rules for handing type abstraction and application:

$$\frac{\Delta, \alpha; \Gamma \vdash t : \tau}{\Delta; \Gamma \vdash t : \forall\alpha.\tau}\ (\mathit{DERtabs}) \qquad \frac{\Delta; \Gamma \vdash t : \forall\alpha.\tau \quad \Delta \vdash \tau_1\ \mathit{type}}{\Delta; \Gamma \vdash t : \tau[\tau_1/\alpha]}\ (\mathit{DERtapp})$$

where *DERtabs* has the side condition that α is not free in Γ.

The approach of directly defining reducibility does not work for System F because we cannot make the argument that the datatype representing reducibility is inductive on the *tp* index. For this reason we need to generalize to reducibility candidates which are all the predicates satisfying CR 1, CR 2 and CR 3. We encode predicates as static terms of sort $tm \to prop$ (we define $rc \equiv tm \to prop$ for convenience) and we define propositions:

$$\begin{aligned}
CR1(R) &\equiv \forall t : tm.\ R(t) \to SN0(t) \\
CR2(R) &\equiv \forall t : tm. \forall t' : tm.\ (R(t), RED0(t, t')) \to R(t') \\
CR3(R) &\equiv \forall t : tm.\ (NEU(t), \forall t' : tm. RED0(t, t') \to R(t')) \to R(t) \\
RC(R) &\equiv (CR1(R), CR2(R), CR3(R))
\end{aligned}$$

Strong normalization (*SN0*) is defined just as before. It is straightforward to show that *SN0* meets the three conditions:

$$sn\_is\_rc : RC(SN0)$$

As a consequence of *CR3*, any reducibility candidate holds for the constant:

$$cr\_cst : \forall R : rc.\ RC(R) \to R(TMcst)$$

The crux of the reducibility candidates is to define interpretations for types as reducibility candidates and to show that whenever a term t can be given a type τ, it is in the reducibility candidate that interprets τ. The fact that a term is strongly normalizing if it is in a reducibility candidate gives us the final result.

In order to interpret types as candidates, we define the arrow and universal quantification constructors for reducibility candidates:

$$\begin{aligned}
RCFUN0(R_1, R_2)(t) &\equiv \forall t_1 : tm.\ R_1(t_1) \to R_2(TMapp(t, t_1)) \\
RCALL0(RF)(t) &\equiv \forall R : rc.\ RC(R) \to (RF(R))(t)
\end{aligned}$$

And we prove that these constructors preserve candidates:

$$\begin{aligned}
rcfun_is_rc &: \forall R_1 : rc. \forall R_2 : rc.\ (RC(R_1), RC(R_2)) \to RC(RCFUN0(R_1, R_2)) \\
rcall_is_rc &: \forall RF : rc \to rc.\ (\forall R : rc. RC(R) \to RC(RF(R))) \to RC(RCALL0(RF))
\end{aligned}$$

It is an important property that the typing rule for lambda is sound with respect to the arrow on candidates:

$$\begin{aligned}
abs_lemma : \forall R_1 : rc. \forall R_2 : rc. \forall f : tm \to tm. \\
(RC(R_1), RC(R_2), \forall t : tm.\ R_1(t) \to R_2(f\ t)) \to RCFUN0(R_1, R_2)(TMlam f)
\end{aligned}$$

To provide a context for parameters in reducibility candidates, we define the sort *rcs* for lists of reducibility candidates:

$$RCSnil : rcs \qquad\qquad RCScons : (rc, rcs) \to rcs$$

In order to lookup parameters in the list we use a datatype (similar to *INCTX*) with type constructor $RCSI : (rcs, rc, int) \to prop$ and term constructors:

$$\begin{aligned}
RCSIone &: \forall R : rc. \forall C : rcs.\ RCSI(RCScons(R, C), R, 0) \\
RCSIshi &: \forall R : rc. \forall R' : rc. \forall C : rcs. \forall n : nat. \\
&\qquad RCSI(C, R, n) \to RCSI(RCScons(R', C), R, n + 1)
\end{aligned}$$

We actually use *rcs* to represent Δ in typing derivations, which have type constructor $DER : (rcs, ctx, tm, tp, int) \to prop$. Only the length of the *rcs* term matters in derivations (the actual predicates in the list are not reflected in the dynamic representation), and derivations with an empty Γ and any Δ are adequately encoded. The use of *rcs* in *DER* (rather than simply a natural number bound on the indices) makes some of the lemmas easier to state.

Next, we define the interpretation of types as reducibility candidates with parameters. For this, we use a dependent datatype with type constructor $TPI : (rcs, tp, rc, int) \to prop$, and term constructors:

$$\begin{aligned}
TPIvar &: \forall C : rcs.\forall T : tp.\forall R : rc.\forall n : nat.\ RCSI(C, R, n) \to TPI(C, TPvar\ n, R, 0) \\
TPIfun &: \forall C : rcs.\forall T_1 : tp.\forall T_2 : tp.\forall R_1 : rc.\forall R_2 : rc.\forall n_1 : nat.\forall n_2 : nat. \\
&\quad (TPI(C, T_1, R_1, n_1), TPI(C, T_2, R_2, n_2)) \to \\
&\quad TPI(C, TPfun(T_1, T_2), RCFUN0(R_1, R_2), n_1 + n_2 + 1) \\
TPIall &: \forall C : rcs.\forall T : tp.\forall RF : rc \to rc.\forall n : nat. \\
&\quad (\forall R : rc.TPI(RCScons(R, C), T, RF(R), n)) \to \\
&\quad TPI(C, TPall(T), RCALL0(RF), n + 1)
\end{aligned}$$

For convenience we define $TPI0(C, T, R) \equiv \exists n : nat.TPI(C, T, R, n)$. In order to prove that the interpretation of a type is a reducibility candidate if all the free variables are interpreted by reducibility candidates, we introduce a datatype $RCS : (rcs, int) \to prop$ such that $RCS(C, n)$ is a sequence of proofs of $RC(R)$ for each R in C. We can then prove the desired lemma:

$$tpi\_is\_rc : \forall C : rcs.\forall T : tp.\forall R : rc.\forall n : nat.\ (RCS0\ C, TPI(C, T, R, n)) \to RC(R)$$

where $RCS0(C) \equiv \exists n : nat.RCS(C, n)$.

The last major lemma we need is a substitution lemma on interpretations of types, which we omit for brevity. In order to state the main lemma, we need to define an environment mapping terms to proofs showing that the terms in the appropriate candidates. For this we use the datatype $ETA : (rcs, ctx, int) \to prop$ where $ETA(C, G, m)$ is a sequence of pairs of $(TPI0(C, T, R), R(t))$ for each (t, T) in G. The main lemma is:

$$\begin{aligned}
der_rc_lemma &: \forall G : ctx.\forall t : tm.\forall T : tp.\forall n : nat.\forall C : rcs.\forall m : nat. \\
&\quad (DER(C, G, t, T, n), ETA(C, G, m), RCS0\ C) \to \\
&\quad \exists R : rc.\ (TPI0(C, T, R), R(t))
\end{aligned}$$

The proof of this lemma is quite involved, mostly due to manipulations of de Bruijn indices. The final theorem is then easy to prove:

$$der\_sn : \forall t : tm.\forall T : tp.DER0(RCSnil, CTXnil, t, T) \to SN0(t)$$

This simply means that every well-typed expression in System F is strongly normalizing.

6 Related Work

There have been several formalizations of proofs of normalization for STLC in the past. Abel [1] encodes a proof of *weak* normalization for STLC in Twelf. As in our proof, the object language is represented using HOAS. How-

ever, normalization is proved using an inductive characterization of the weakly normalizing terms, following Joachimski and Matthes [8], rather than Tait's method of reducibility predicates. Sarnat and Schürmann [11] have recently given a proof of weak normalization directly in Twelf using a logical relation. They encode minimal first-order logic which is then used in the definition of the logical relation. It is not clear whether their technique would allow a similar encoding of strong normalization. Berger, Berghofer, Letouzy and Schwichtenberg [3] give proofs of strong normalization for STLC using Tait's method in three systems: Isabelle/HOL, Coq, and Minilog. They also analyze the programs that can be extracted from the formal proofs. However, the formalizations described all make use of first-order representations (using either de Bruijn indices or names for variables) rather than HOAS and also start from a large number of unproven axioms (eleven).

Strong normalization for System F has previously been formalized by Altenkirch [2] using the Lego system. His formalization uses the de Bruijn encoding for both terms and types, and because of this, is significantly longer and more complicated than our proof. Even though our formalization contains full proof terms, rather than tactic-based scripts, it is shorter by about a factor of two.

7 Conclusion

We have presented formalizations of proofs of strong normalization for STLC and System F which use HOAS and Tait's and Girard's methods (respectively). The unique features of ATS/LF (in particular the separation between statics and dynamics) allow for the encoding of powerful logical relations arguments over the simple and elegant language encodings enabled by HOAS. In these proofs we found that HOAS made it much easier to deal with the mundane details of naming and substitution, which often take the majority of the effort in first-order encoding. [4] As a result, we are able to define the syntax and semantics of STLC and prove strong normalization as described, all in less than 300 lines of commented ATS/LF code! For System F, the proof is likewise short, under 900 lines.

References

[1] Abel, A., *Weak normalization for the simply-typed lambda-calculus in Twelf*, in: *Logical Frameworks and Metalanguages (LFM 04)*, IJCAR, Cork, Ireland, 2004.

[2] Altenkirch, T., *A Formalization of the Strong Normalization Proof for System F in LEGO*, in: M. Bezem and J. F. Groote, editors, *Proceedings of the*

[4] Actually, we have also formalized a strong normalization proof of STLC that uses FOAS to represent lambda-terms:

`http://www.cs.bu.edu/~hwxi/ATS/EXAMPLE/LF/STLC-SN-foas.dats`

There are several unproven lemmas in this formalization, which can certainly be finished but require some effort on handling substitution that is uninspiring and tedious.

International Conference on Typed Lambda Calculi and Applications (1993), pp. 13–28.

[3] Berger, U., S. Berghofer, P. Letouzey and H. Schwichtenberg, *Program extraction from normalization proofs*, Studia Logica (2005), special issue, to appear.

[4] Chen, C. and H. Xi, *Combining Programming with Theorem Proving*, in: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, Tallinn, Estonia, 2005, pp. 66–77.

[5] Donnelly, K. and H. Xi, *Combining higher-order abstract syntax with first-order abstract syntax in ATS*, in: *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding* (2005), pp. 58–63.

[6] Girard, J.-Y., *Une Extension de l'Interprétation de Gödel à l'Analyse, et son Application à l'Élimination des Coupures dans l'Analyse et la Théorie des Types*, in: J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, Studies in Logic and the Foundations of Mathematics **63** (1971), pp. 63–92.

[7] Girard, J.-Y., Y. Lafont and P. Taylor, "Proofs and Types," Cambridge Tracts in Theoretical Computer Science **7**, Cambridge University Press, Cambridge, England, 1989, xi+176 pp.

[8] Joachimski, F. and R. Matthes, *Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T*, Arch. Math. Log. **42** (2003), pp. 59–87.

[9] Pfenning, F. and C. Elliott, *Higher-order abstract syntax*, in: *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, Atlanta, Georgia, 1988, pp. 199–208.

[10] 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)* (1999), pp. 202–206.

[11] Sarnat, J. and C. Schürmann, *On the Representation of Logical Relations*, Yale University Technical Report, YaleU/DCS/TR1362 (2006).

[12] Tait, W. W., *Intensional Interpretations of Functionals of Finite Type I*, Journal of Symbolic Logic **32** (1967), pp. 198–212.

[13] Xi, H., *Dependent Types for Program Termination Verification*, Journal of Higher-Order and Symbolic Computation **15** (2002), pp. 91–132.

[14] Xi, H., *Dependently Typed Pattern Matching*, Journal of Universal Computer Science **9** (2003), pp. 851–872.

[15] Xi, H., *Applied Type System* (2005), available at: `http://www.cs.bu.edu/~hwxi/ATS`.

A ATS/LF proof of final lemmas and theorem

```
...

// application reducibility lemma
prfun reduceFun
  {f:tm->tm, t:tm, T1:tp, T2:tp, n1:nat, n2:nat} .<n1+n2>.
  (tp1: TP0 T1, tp2: TP0 T2,
   sn1: SN(TMlam f, n1), sn2:SN(t, n2), r1:R(t, T1),
   fr2: {t:tm} R(t, T1) -> R(f t, T2)): R(TMapp(TMlam f, t), T2) = let

   prval r1' = fr2 r1
   prfn fr {t':tm} (red:RED0(TMapp(TMlam f, t), t')) : R(t', T2) = case* red of
     | REDapp1(red') =>
       let
          prval REDlam {f, f',_} fred' = red'
          prfn fr2' {t:tm} (r: R(t, T1)): R(f' t, T2) =
            cr2(tp2, fr2 r, fred'{t})
       in
          reduceFun(tp1, tp2, forwardSN(sn1, red'), sn2, r1, fr2')
       end
     | REDapp2(red') =>
       reduceFun(tp1, tp2, sn1, forwardSN(sn2, red'), cr2(tp1, r1, red'), fr2)
     | REDapp3() => r1'
in
   cr3(NEUapp, tp2, fr)
end

// the abstraction rule is sound with respect to redicible terms
prfn absSound {f:tm->tm, T1:tp, T2:tp}
  (tp1: TP0 T1, tp2: TP0 T2,
   frr : {t:tm} R(t, T1) -> R(f t, T2)) : R(TMlam f, TPfun(T1, T2)) =
  let
     prfn fr {t:tm} (rt: R(t, T1)) : R(TMapp(TMlam f, t), T2) =
       let
          prval snt = cr1(tp1, rt)
          prval snf = lamSN(cr1 (tp2, frr {TMcst} (cr4 tp1)))
       in
          reduceFun (tp1, tp2, snf, snt, rt, frr)
       end
  in
     Rfun(fr)
  end

// pick specified reducibility predicate from the sequence
prfun rGet {t:tm, T:tp, G:ctx, n:nat} .<n>.
  (i:INCTX(t,T,G,n),rs: RS0(G)) : R(t,T) = case* i of
  | INCTXone() => (case* rs of RScons(r,_) => r)
  | INCTXshi i => (case* rs of RScons(_,rs) => rGet(i, rs))

// The assigned type can be extracted from a derivation
prfun der2tp {G:ctx, t:tm, T:tp, n:nat} .<n>. (der: DER(G,t,T,n)): TP0 T =
  case* der of
    | DERvar (_, tp) => tp
    | DERlam (tp1, derf) => let prval tp2 = der2tp derf in TPfun (tp1,tp2) end
    | DERapp (der1, der2) => let prval TPfun (_, tp2) = der2tp der1 in tp2 end

// main lemma
```

```
prfun reduceLemma {G:ctx, t:tm, T:tp, n:nat} .<n>.
  (der: DER(G,t,T,n), rs: RS0 G): R (t, T) =
  case* der of
    | DERvar (i,_) => rGet (i, rs)
    | DERlam {_,f,T1,T2,_} (_, derf) =>
      let
         prval TPfun{T1, T2, s1, s2} (tp1, tp2) = der2tp der
         prfn gr {t:tm} (r: R(t,T1)): R(f t, T2) = let
           prval rs' = RScons (r, rs)
           prval r' = reduceLemma (derf{t}, rs')
         in
           r'
         end
         prfn fr {t:tm} (r: R(t,T1))
           : R(TMapp(TMlam f, t), T2) = let
           prval lamf_red = absSound(tp1, tp2, gr)
           prval Rfun(red_imp) = lamf_red
         in
           red_imp r
         end
      in
         Rfun fr
      end
    | DERapp (der1, der2) =>
      let
         prval r1 = reduceLemma(der1, rs)
         prval Rfun fr = r1
         prval r2 = reduceLemma(der2, rs)
      in
         fr r2
      end

// all typable terms are reducible
prfn reduce {t:tm, T:tp} (der: DER0 (CTXnil,t,T)): R (t,T) =
  reduceLemma(der, RSnil())

// the final theorem
prfn normalize {t:tm, T:tp} (der: DER0 (CTXnil,t,T)): SN0 t =
  cr1(der2tp der, reduce der)
```