Some Normalization Properties of Martin-Löf's Type Theory, and Applications

David A. Basin*
Department of Artificial Intelligence
University of Edinburgh
Edinburgh EH1 1HN, Scotland
basin@aipna.ed.ac.uk

Douglas J. Howe†
Department of Computer Science
Cornell University
Ithaca, NY 14853, U.S.A.
howe@cs.cornell.edu

Abstract

For certain kinds of applications of type theories, the faithfulness of formalization in the theory depends on intensional, or structural, properties of objects constructed in the theory. For type theories such as LF, such properties can be established via an analysis of normal forms and types. In type theories such as Nuprl or Martin-Löf's polymorphic type theory, which are much more expressive than LF, the underlying programming language is essentially untyped, and terms proved to be in types do not necessarily have normal forms. Nevertheless, it is possible to show that for Martin-Löf's type theory, and a large class of extensions of it, a sufficient kind of normalization property does in fact hold in certain well-behaved subtheories. Applications of our results include the use of the type theory as a logical framework in the manner of LF, and an extension of the *proofs-as-programs* paradigm to the synthesis of verified computer hardware. For the latter application we point out some advantages to be gained by working in a more expressive type theory.

1 Introduction

For certain kinds of applications of type theories, the faithfulness of formalization in the theory depends on intensional, or structural, properties of objects constructed in the theory. One example of this is the use of a typed lambda calculus, such as LF [10], as a "general logic". One can embed a natural deduction logic in the calculus by introducing typed constants for each rule and for each term and formula constructor of the logic. The constants are typed in such a way that for each sentence P of the logic there is a type T whose members correspond to proofs of the sentence. The encoding is faithful when the existence of a member of T implies the provability of P. To prove faithfulness, one analyzes the normal forms of members of T and observes that their construction from the constants is isomorphic to a proof. See [10] for a discussion of the advantages of the LF approach to encoding logics.

Another example is a widely used method for formalizing the combinational logic of digital circuits [4]. Here one represents a circuit as a relation over the booleans, so that a circuit with n ports (inputs and outputs) is represented as an n-ary relation which is true exactly when the ports have values consistent with the logic of the circuit. The relations are built from operators which express the logic of transistors, gates, registers, etc. If *bool* is the type of booleans, then a circuit with 2 ports could be represented as a function of type $bool \times bool \to bool$. If we can form subtypes in our type theory, then we might write

$$\{\, R : bool \times bool \to bool \mid Spec\,(R)\,\}$$

*Supported by NATO under a grant awarded in 1990.
†Supported in part by ONR contract N00014-88-K-0409.

to represent the collection of all circuits satisfying some specification. A term t in this type is a function computing boolean values as specified. However, we need to know more than this if we want our representation to be faithful. In particular, we need to know that t is built from the operators for transistors, etc., in such a way that it can be viewed as a circuit. Here again we would like to analyze normal forms.

In this paper we show how the kinds of encodings just discussed can be done directly and faithfully in Martin-Löf's polymorphic type theory, and also in a class of "computational" extensions of it.

Martin-Löf's polymorphic type theory, which we will henceforth refer to as TT, was designed as a formal basis for constructive mathematics. It contains a functional programming language and a highly expressive type system. Some of its descendents have been used in a number of interactive proof development systems (for example, Nuprl [6], Veritas [8], Isabelle [16] and a system based on [15] currently under development). Nuprl, in particular, has been extensively used to formalize mathematics and programming. Our results should apply, with minor modifications, to the type theory of Nuprl, but this has not been checked.

The difficulty in establishing the required normalization properties in TT is that TT gains its expressiveness and flexibility partly at the cost of some of the traditional properties of type systems. In particular, the underlying programming language of TT is untyped, and terms that belong to types cannot, in general, be normalized in the usual λ-calculus sense (although they are guaranteed to evaluate to weak head-normal form). For example, if $\perp$ is any diverging term of the untyped λ-calculus and T is an empty type, then

$$\lambda x.\ \perp\ \in T \rightarrow T.$$

Thus the techniques for proving faithfulness in LF are not directly applicable to TT.

Our main result is that a sufficient kind of normalization property does in fact hold for TT. From this property, we can derive several useful results about the faithfulness of encodings in TT. Before describing these results, we consider an example.

Suppose we have proven the sequent

$$T\!:\!U_1,\ z\!:\!T,\ s\!:\!T \rightarrow T,\ r\!:\!Ax \vdash t \in T$$

in TT, where r does not occur free in t and where Ax is the encoding in TT, via the *propositions-as-types* correspondence, of some logical statement involving T, z and s. This sequent says that under the hypotheses that T, z and s have the given types and satisfy Ax, t has type T (U_1 can be thought of as the collection of all "small" types).

Since T is a hypothetical type, and the only operations given for constructing members of T are z and s, we expect that t should be equivalent to a program of the form $s(s(\ldots(z)\ldots))$. There is little hope of proving such a property by a metatheoretic analysis of TT's inference system. The rules of this type theory are too numerous and complex.

Instead, we take a semantic approach, using the semantics developed by Allen [1]. In this semantics, the meaning of a type is a set of closed terms (together with an equivalence relation over the set). A term belongs in a type if and only if it evaluates to a value of the right form. For example, the members of N are all terms that evaluate to a numeral.

To obtain a normal form for t in the above example, we might proceed as follows. Since the sequent is provable, it is true. Its truth implies that for any particular type T, and for any values $z \in T$ and $s \in T \rightarrow T$ such that Ax is satisfied, the term t must evaluate to a value of a form appropriate for T. If such T, z, and s exist, then there is a sequence of computation steps taking this instance of t to a value. Now consider applying the same sequence to t, a term which may have free occurrences of the *variables* T, z and s. We get a sequence of "symbolic computation" steps that must also terminate, although it might be shorter than the original sequence. Since the computation system is lazy, the result of this new sequence may only be partly normalized, so we may need to repeat the whole process with some of the subterms.

However, we have the following difficulty. Suppose, for example, that Ax asserts both that T is isomorphic to the type of natural numbers and that s solves the halting problem for Turing machines. Since all members of function types in the semantics are computable, no such s can exist. Hence there are no terms T, z, s, r of TT for which the hypotheses are true. Thus the sequent is vacuously true for any t whatsoever, in particular for $t = \perp$. Nevertheless, it would clearly be useful to use the type theory to reason about such apparently inconsistent contexts, and still be guaranteed a normalization property.

This difficulty can in fact be ruled out because TT is *open-ended* in the sense that its theorems are true in any semantics constructed from a collection of new base types and a computation language that extends that of TT. For the example above, we could use an extension of TT that includes an "oracle" deciding the halting problem.

More generally, we want to show that if

$$x_1 : A_1, \ \ldots, \ x_n : A_n, \ r : Ax \vdash t \in T$$

is provable in TT, where $T, A_1, \ldots, A_n$ are from a well-behaved subtheory of TT and t is an arbitrary term in which r does not occur free, and if there is some extension of TT in which we can satisfy the hypotheses, then we can effectively find a term t in the subtheory which is in normal form and which is equal to t (as a member of T). The subtheory we deal with in this paper is based on the LF type theory. It is suitably well-behaved and is expressive enough for the examples we have in mind. We also fix a particular "maximal" extension in which to satisfy the hypotheses. This extension is a full classical model of TT in which all function types have representatives of all functions with appropriate domain and range.

The core of the paper is a pair of fairly general technical lemmas. We use these lemmas to obtain several particular theorems which can be applied to the practical examples outlined above. Suppose

$$\Gamma, \ r : Ax \vdash t \in T$$

is provable in TT, where

$$\Gamma = x_1 : A_1, \ \ldots, \ x_n : A_n$$

is the translation into TT of an LF context, T is an LF type in this context, t is any term of TT in which r is not free, and Ax represents a formula of higher-order logic in which the types of quantified variables are the LF types well-formed in Γ. We show that either of the following conditions is sufficient to find an LF term t' in normal form such that $\Gamma \vdash t' \in T$ in LF and

$$\Gamma, \ r : Ax \vdash t = t' \in T$$

is true in TT:

1. Ax is the trivial proposition *true*.
2. All the types well-formed in Γ are simple types (built from $\rightarrow$ and base types), and the hypotheses can be satisfied (in a certain way) in the classical extension.

These conditions cover many cases of practical interest, including the use of TT as a logical framework (where Ax is *true*) and the use of TT as a foundation for reasoning about combinational logic (where Ax expresses the logic properties of the circuit constructors). It should be possible to obtain more general results along these lines within the same technical framework.

The classical model is a reasonable place to look for objects to satisfy hypotheses with. However, it also plays a crucial technical role. The functions and values that are introduced in the classical extension have a simple structure. The proofs of the technical lemmas rely heavily on this simple structure, and we do not know how to eliminate this reliance.

In the open-ended spirit of TT, we prove the above results not just for TT *per se*, but for any extension of it which has the same type constructors but a larger computation language. We assume that any such language is presented with a certain kind of structural operational semantics. The proofs of the technical lemmas involve a detailed analysis of these presentations. The technical work of this paper builds on the work in [13], which introduces the operational semantics formalism and constructs the classical extension of TT.

There are some advantages to being able to use a highly expressive type theory for the practical applications mentioned above. For example, within stronger type theories, the type theory itself can serve as a meta-language for developing derived inference rules and tactics[11]; this is not possible within the weaker LF type theory. Another benefit to using a rich type theory is that one can reason about parameterized terms where the parameters come from arbitrary (not necessarily LF) types. For example, in the domain of hardware verification and synthesis, it is best to verify large regular circuits such as adders and multipliers by verifying parameterized versions (e.g., an n-bit adder) and later instantiating the parameters to produce a term representing a specific circuit. We provide an example of this kind of development at the end of this paper.

Some closely related work is presented in [5]. The main result there is that LF-like encodings in Nuprl are faithful in the case where the encoded logic has a complete semantics that can be introduced into the Nuprl semantics. The notion of faithfulness used there is weaker, since it only guarantees that provability in the object logic is equivalent to provability in the encoding, and does not guarantee any structural correspondence between proofs and terms of the type theory. Furthermore, the only proof given is for encodings of first-order logic.

The remainder of the paper is organized as follows. In the next section we supply some of the necessary background, summarize the relevant sections of [12, 13] and briefly describe the semantic approach of [1]. We then state and sketch the proofs of the results referred to above. In the last section we describe some applications.

2 Background

2.1 Structured Computation Systems

Language

A *lazy computation language* L is a triple (O, K, α) where O and K are sets of *operators* with $K \subset O$ and

$$\alpha \in O \to \{\, (k_1, \ldots, k_n) \mid n, k_i \geq 0 \,\}.$$

We call the members of K *canonical* operators. For $\tau \in O$, $\alpha(\tau)$ is the *arity* of τ and specifies the number and binding structure of the operator's arguments.

Fix a countably infinite set of variables. We inductively define the *second-order terms of arity n*. A variable is a second-order term of arity 0. If τ is an operator with arity $(k_1, \ldots, k_n)$, if $b_1, \ldots, b_n$ are second-order terms of arity $k_1, \ldots, k_n$ respectively, and if $\overline{x}$ is a list of m variables, then then $\overline{x}.\, \tau(b_1, \ldots, b_n)$ is a second-order term of arity m. The b_i in this term are the *operands* of the operator τ. We add binding structure by specifying that in a second-order term $x_1, \ldots, x_m.t$, each x_i binds in t. A *term* is a second-order term of arity 0.

We identify second-order terms that are α-equal (the same up to renaming of bound variables). We write $t[a_1, \ldots, a_n/x_1, \ldots, x_n]$ for the result of simultaneous substituting in t the terms $a_1, \ldots, a_n$ for $x_1, \ldots, x_n$, respectively. We will simplify this to $t[a_1, \ldots, a_n]$ when the variable list can be inferred from the context.

A term $\tau(\overline{t})$ is *canonical* if τ is a canonical operator, and *noncanonical* otherwise. The closed terms will be the programs of our computation systems. The closed canonical terms will be exactly the results of evaluating programs; thus a term will be considered to be fully evaluated exactly if its outermost operator is canonical.

Inference rules for evaluation, described below, are specified using an extension of the set of terms. For each $i \geq 1$ we fix an infinite set of variables which we will call the *second-order variables of arity i*. The ordinary variables from above are the second-order variables of arity 0. A *term schema* is built in the same way as a term, except that we also include expressions of the form $P[\overline{t}]$ where P is a second-order variable of arity i ($i \geq 0$) and $\overline{t}$ is a list of i terms. We identify $P[]$ and P. Because of the use we will make of second-order variables, we will usually capitalize the free second-order variables of a term schema and refer to them as *metavariables* (note that variables of arity $i > 0$ are always free).

A second-order substitution is a partial map σ from second-order variables to second-order terms of corresponding arity. The definition of the application of σ to terms is similar to the definition for ordinary substitution, except that $\sigma(P[a_1, \ldots, a_n]) = t[\sigma(a_1), \ldots, \sigma(a_n)/x_1, \ldots, x_n]$ if $\sigma(P)$ is $x_1, \ldots, x_n.t$. A *simple term schema* has the form $\tau(\overline{x}_1.\,P_1[\overline{x}_1]; \ldots; \overline{x}_n.\,P_n[\overline{x}_n])$ where the P_i are distinct metavariables. It is (non-) canonical if τ is (non-) canonical.

Evaluation Rules

Let L be a lazy computation language. An *evaluation rule* for L is a rule whose premises and conclusion are of the form $a \Downarrow b$, where a and b are term schemas or metavariables of arity 0. The set of premises of a rule may be infinite but must be well-ordered (that is, ordered by some total well-founded relation). In addition, in order to guarantee certain properties of evaluation, we impose the following syntactic conditions on a rule. Let I be the set that well-orders the premises, and for $i \in I$, let $a_i \Downarrow b_i$ be the i^{th} premise. Let $a \Downarrow b$ be the conclusion of the rule. We require the following: (1) a is a noncanonical simple term schema; (2) for all $i \in I$, b_i is a metavariable or a canonical simple term schema, and has no metavariables in common with a or with b_j for $j \neq i$; (3) for each $i \in I$ and metavariable P of a_i, P occurs in a or in b_j for some $j <_I i$; (4) b is a metavariable, and for some i, b is b_i and does not occur in any other premise. The metavariable b is called the *output variable*, and the i^{th} premise in the last condition is called the *output premise*.

The last restriction is only a convenience: rules allowing a more general form for b can be modified to meet this restriction by adding a few premises. For example, a rule with no output premise and with conclusion $a \Downarrow v$, for some canonical v, can be taken as shorthand for a rule with output premise $v \Downarrow P$ and conclusion $a \Downarrow P$.

An *instance* of a rule is the result of applying to its premises and conclusion a second-order substitution whose domain is the set of metavariables occurring in the rule. An instance is closed if each formula is closed (i.e. both terms are closed).

Let R be a set of evaluation rules over a lazy computation language L. We inductively define, for terms a and b, *derivations (in R) of $a \Downarrow b$*. If a is canonical then $a \Downarrow a$ is a derivation of itself. If a is noncanonical then a derivation of $a \Downarrow b$ consists of a rule instance with conclusion $a \Downarrow b$ and premises $\{ a_i \Downarrow b_i \mid i \in I \}$, together with derivations of $a_i \Downarrow b_i$ for each $i \in I$. A derivation is *closed* if every rule instance in it is closed. When we write $a \Downarrow b$ we will mean that the formula $a \Downarrow b$ has closed derivation.

A *structured computation system* S consists of a lazy computation language L and a set R of evaluation rules over L. The relation $\cdot \Downarrow \cdot$ specified by R is the *evaluation relation* of S. The evaluation relation has the property that if $a \Downarrow b$ then b is canonical, and if a is canonical then $a \Downarrow b$ if and only if $b = a$. A closed term b is a *value* of a if $a \Downarrow b$. A closed formula $a \Downarrow b$ is *determinate* if it is true and if for all closed terms v, $a \Downarrow v$ implies $v = b$. S is *deterministic* if every true closed formula over S is determinate. S is *strongly deterministic* if it is deterministic and if for every closed instance r of a rule of S, if the conclusion of r is

$a \Downarrow b$ and its premises are $\{ a_i \Downarrow b_i \mid i \in I \}$, and if there is a $j \in I$ such that a_j has no value and $a_i \Downarrow b_i$ is determinate for $i < j$, then a has no value.

For example, the lazy untyped λ-calculus can be cast as a structured computation system. Take $O = \{ \lambda, ap \}$ and $K = \{ \lambda \}$. Define α by $\alpha(\lambda) = (1)$ and $\alpha(ap) = (0, 0)$. Write $\lambda x.\, b$ for $\lambda(x.\, b)$, and $a(b)$ for $ap(a, b)$. The single evaluation rule is

$$\frac{F \Downarrow \lambda x.\, B[x] \quad B[A] \Downarrow C}{F(A) \Downarrow C.}$$

Languages with eager evaluation can also be cast as lazy computation systems. For example, we can specify the call-by-value λ-calculus by changing the above rule to

$$\frac{F \Downarrow \lambda x.\, B[x] \quad A \Downarrow V \quad B[V] \Downarrow C}{F(A) \Downarrow C.}$$

Non-lazy data constructors can be dealt with by introducing a non-canonical operator for each constructor. For example, for a pairing operator $\langle \cdot, \cdot \rangle$, introduce a non-canonical operator *pair* with the following rule.

$$\frac{A \Downarrow V \quad A' \Downarrow V'}{pair(A; A') \Downarrow \langle V, V' \rangle}$$

If S and S' are structured computation systems, then S' *extends* S if the language of S' extends that of S, and if for every rule r of S', if the operator in the conclusion of r is in the language of S then r is in S (so S' does not extend the semantics of any of the operators of S).

Some of the results below state that certain terms can be found *effectively*. For this to be true, it is required that the structured computation system be effectively presented.

A Computational Preorder

Let S be a structured computation system. The preorder $\leq$ is defined as the largest binary relation over the closed terms of S such that $a \leq a'$ if and only if: $a \Downarrow \theta(\overline{x}_1.\, t_1; \ldots; \overline{x}_n.\, t_n)$, for θ canonical, implies there are terms $t'_1, \ldots, t'_n$ such that $a' \Downarrow \theta(\overline{x}_1.\, t'_1; \ldots; \overline{x}_n.\, t'_n)$ and such that for each i, $1 \leq i \leq n$ and every sequence of closed terms $\overline{a}$ of the appropriate length, $t[\overline{a}/\overline{x}] \leq t'[\overline{a}/\overline{x}]$. Define $a \leq b$, for a and b open terms, if $\sigma(a) \leq \sigma(b)$ for every substitution σ such that $\sigma(a)$ and $\sigma(b)$ are closed. Finally, define $a \sim b$ if $a \leq b$ and $b \leq a$.

Following are some examples related to the computation system of TT. $\langle 2+2, 3 \rangle \sim \langle 4, 3 \rangle$. $\lambda x.\, x + 2 \sim \lambda x.\, 2 + x$. $y \not\sim \lambda x.\, y(x)$. $(\lambda x.\, b)(a) \sim b[a/x]$. Finally, for the pure untyped λ-calculus, the following are equivalent for closed terms a and a': (1) $a \sim a'$; (2) for all $n \geq 0$ and closed $b_1, \ldots, b_n$, $ab_1 \ldots b_n$ has a value if and only if $a'b_1 \ldots b_n$ does; (3) for every context (*i.e.* term with a hole) $C[\cdot]$, $C[a]$ has a value if and only if $C[a']$ does.

In [13] we prove the following.

Theorem 1 *The preorder $\leq$ of S is a congruence:*

$$\tau(\overline{x}_1.\, t_1; \ldots; \overline{x}_n.\, t_n) \leq \tau(\overline{x}_1.\, t'_1; \ldots; \overline{x}_n.\, t'_n)$$

whenever $t_i \leq t'_i$ for each i. If follows that if $a \leq a'$ and $b \leq b'$ then $b[a/x] \leq b'[a'/x]$.

A trivial consequence is that these properties also hold of $\sim$.

2.2 Type Theory

Let S_{TT} be the computation system of TT [14]. It is trivial to cast S_{TT} as a structured computation system.

The inductive construction in [1] gives a semantics of TT as a *type system*, which is a partial function ϕ mapping closed terms to partial equivalence relations[1] over closed terms. If $\phi(T)$ is defined we say T is a type in ϕ. For T a type in ϕ, we write $t \in_\phi T$ when $(t,t) \in \phi(T)$, and $t = t' \in_\phi T$ when $(t,t') \in \phi(T)$. The terms t such that $t \in_\phi T$ are the *members* of T. Two types are *equal* if they are assigned the same partial equivalence relation.

A slightly more complicated construction gives a semantics for Nuprl [6]. The main difference is that Nuprl has an intensional, or structural, type equality.

Theorem 2 *The rules of TT and Nuprl are "computationally open-ended", in the sense that for any deterministic S which extends S_{TT}, the construction in [1] yields a type system ϕ_S in which the rules of TT are true.*

We will refer to such an S as a *model* of TT, and will write $t \in_S T$ in place of $t \in_{\phi_S} T$, and similarly for $t = t' \in_S T$.

The rules of TT deal with judgements under hypotheses. The inference system can be given a sequent formulation. In this paper we will only need to mention sequents of the form

$$x_1 : A_1, \ldots, x_n : A_n \ \vdash C$$

where C is of the form $t \in T$ or $t = t' \in T$. We will use TT as a subscript, as in

$$x_1 : A_1, \ldots, x_n : A_n \ \vdash_{TT} C,$$

when the sequent is provable in TT. The semantics of the type theory can be extended to a semantics of sequents, although this is somewhat complicated. We will use S as a subscript when a sequent is true in the semantics generated from S.

There are a number of type constructors in TT. As examples, we consider Π types and universes. The term $\Pi\, x \in A \,.\, B$ is a type exactly if A is a type and for every $a \in A$, $B[a/x]$ is a type (we will ignore equality—for example, in this case we should also require that the type family B respect the equality of A). Its members are the terms t where $t \Downarrow \lambda x.\, b$ for some b such that $b[a/x] \in B[a/x]$ for every $a \in A$. Any term T such that $T \Downarrow \Pi\, x \in A \,.\, B$ is a type which is equal as a type to $\Pi\, x \in A \,.\, B$. For each i there is a type U_i whose members are types and which is closed under all the type constructors except U_j for $j \geq i$.

We will need the following result from [12].

Theorem 3 *Let S be a model of TT and let T be a type in S. If $T \leq_S T'$ then T' is a type which is equal to T. If $t \in_S T$ and $t \leq_S t'$ then $t = t' \in_S T$.*

TT is also open-ended with respect to universe inhabitation. This property is rather difficult to characterize (see [2]), but a weak form is sufficient here. In particular, we can incorporate an arbitrary collection of *base types* meeting the following restrictions. Let S be a model of TT, and let ϕ_S be the type system generated from S. Define a *constant* to be a canonical term with no subterms (that is, it is a canonical operator of arity ()). Let X be a set of constants which are not in the domain of ϕ_S, and let ϕ' be a type system over S whose domain is X and such that for every $c \in X$, if $t = t' \in_{\phi'} c$ then there exists a constant c' such that $t \Downarrow c'$ and $t' \Downarrow c'$. Then ϕ' can be extended to a type system for TT in which all the new base types are members of U_1.

Although the results below apply to an arbitrary S, they refer to provability only in TT. These results are actually more general, since the only property of provability used is that it implies truth in all extensions of S. Thus we could extend TT with new rules as long as they are true in all extensions of S.

[1] A *partial equivalence relation* is a symmetric, transitive relation.

2.3 Classical Models of Type Theory

Let S be a deterministic model of TT. In [13] we show how to extend S to a "classical" model S_c obtained by injecting enough oracles into S to ensure "full inhabitation" of function types in the following sense. Let A and B be types in S_c and let $a \mapsto b_a$ be any *function*, in the usual mathematical sense, mapping members of A to members of B, such that for all members a and a' of A, if $a = a' \in_{S_c} A$ then $b_a = b_{a'} \in_{S_c} B$. Then there is a *term* $f \in_{S_c} A \to B$ such that for every $a \in_{S_c} A$, $f(a) = b_a \in_{S_c} B$. The obvious generalization of this property to Π types holds as well. It follows that the propositions-as-types expression of the law of the excluded middle is sound in S_c (although we will not need this fact). In order to guarantee that S_c is deterministic we actually need to assume slightly more about S then that it is itself deterministic. See [13] for details on this complication (which does not appear to rule out any interesting computation systems).

In this section we present only the parts of the construction of S_c that are required for the subsequent development. We start by choosing a large chunk V of the cumulative hierarchy of set theory. V can be thought of as simply a very large set with $\emptyset$ as the only "primitive" element. To be more precise, define V_α, for each ordinal α, by transfinite induction:

$$V_\alpha = P(\bigcup_{\beta<\alpha} V_\beta).$$

where $P(S)$ is the power set of S. Set $V = V_\alpha$ for some suitably large limit ordinal.

The language of S_c is obtained from that of S by adding, for each n-ary function f $(n \geq 0)$ which as a set is a member of V, a noncanonical operator o_f, called an *oracle*, whose arity is $(0, \ldots, 0)$. For use in the normalization results below, we also add, for each $v \in V$, a canonical operator κ_v, called an *atom*, of arity (). We can define an encoding i of the set of terms of S_c in V, using the standard set-theoretic encodings of pairing, integers etc.

In order to guarantee that S_c be deterministic, we define evaluation rules for an operator o_f only when f satisfies a certain "minimality" property, the details of which are of no concern here. The main property of evaluation we will need is the following.

Lemma 1 *Let $f \in V$ be a minimal function and let $t_1, \ldots, t_n$ be closed terms.*

$$o_f(t_1; \ldots; t_n) \Downarrow c$$

if and only if there exists $(i(u_1), \ldots, i(u_n))$ in the domain of f such that

$$f(i(u_1), \ldots, i(u_n)) = c$$

and for each i, $u_i \leq t_i$.

Henceforth we will identify a term with its code in V.

We will need a large collection of base types in order to provide instantiations of LF contexts. Let A be a large set of atoms (say the set of all κ_v for $v \in V_\alpha$, where α is an inaccessible cardinal number[2]). We introduce base types, as described in the previous section, one for every partial equivalence relation over A. Pick arbitrary atoms to name these base types. Henceforth, we will assume that the type system corresponding to S_c includes these base types.

[2] This is absurdly large for our purposes, but the construction of S_c already uses an infinite sequence of inaccessible cardinals.

3 A Normalization Lemma

In this section we prove a normalization property for open terms t that have a value in the classical semantics when closed terms from a certain class are substituted for free variables. We will call the members of this class of substitutable terms "LF-like" since they correspond to the functions which can inhabit types or kinds in a set theoretic semantics of LF.

The normalization lemma is rather technical. Its significance is mainly as a tool for proving the results of the next section. It is proved with an involved (and somewhat nasty) syntactic argument.

Let V, S and S_c be as in the previous section, except that we additionally require that S be strongly deterministic. We first define a suitably well-behaved class of terms of S_c. We inductively define sets F_n, $n \geq 0$. $F_0 = \{\, \kappa_v \mid v \in V \,\}$. $\lambda x.\, o_f(x) \in F_{n+1}$ if $f \neq \emptyset$ and there exists $k \geq 0$ such that $u \in F_k$ and $v \in F_n$ whenever $(u, v) \in f$. We call the members of F the *LF-like terms*, and the members of F_n the *LF-like terms of arity n*. We will often treat a function $f \in V$ as an LF-like term by identifying it with $\lambda x.\, o_f(x)$. Also, we will sometimes consider κ_v to be an LF-like function of arity 0. An LF-like term of arity n can be thought of as a curried function taking n arguments, which must be LF-like terms, and returning an atom.

We now define a notion of *domain* for the LF-like terms (and functions). For $t \in F$,

$$D_t = \begin{cases} () & \text{if } t \text{ has arity } 0 \\ \{\, (u_1, \ldots, u_n) \mid u_1 \in \mathit{dom}(f) \;\&\; (u_2, \ldots, u_n) \in D_{f(u)} \,\} & \text{if } t = \lambda x.\, o_f(x) \end{cases}$$

Next, we define the application of an LF-like term to a member of its domain.

$$\begin{aligned} \mathit{val}(\kappa_v, ()) &= \kappa_v \\ \mathit{val}(\lambda x.\, o_f(x), (u_1, \ldots, u_n)) &= \mathit{val}(f(u_1), (u_2, \ldots, u_n)) \end{aligned}$$

The following holds: for $t \in F_n$, $t a_1 \ldots a_n \Downarrow c$ if and only if there exists $(u_1, \ldots, u_n) \in D_t$ such that $\mathit{val}(t, \overline{u}) = c$ and for each i, $u_i \leq a_i$.

For technical reasons, we also need a notion of "full η-expansion" of an LF-like term, analogous to the $\beta\eta$-long normal form of the typed λ-calculus. Instead of using types to determine what this expansion is, we us the arity information contained in members of F. First, for t a term and $u \in F_n$, define

$$\eta(t, u) = \begin{cases} t & \text{if } n = 0 \\ \lambda x_1 \ldots x_n.\, t a_1 \ldots a_n & \text{if } n > 0 \end{cases}$$

where, in the second case, none of the variables in $\overline{x}$ are free in t, and $a_i = \eta(x_i, v_i)$ for some arbitrary $\overline{v} \in D_u$. Now define, for $u \in F_n$,

$$u^\eta = \begin{cases} u & \text{if } n = 0 \\ \lambda x_1 \ldots x_n.\, o_f(a_1) a_2 \ldots a_n & \text{if } u = \lambda x.\, o_f(x) \end{cases}$$

where, in the second case, the a_i are as before.

We will use LF-like terms to give meaning to the free variables of a term proven well-typed in some LF context. An *oracle assignment* is a mapping ξ from a finite set of variables to the set of LF-like functions of V. For the same technical reasons as were referred to above, we need to restrict the form of terms to which we can apply ξ. Let t be a term such that for every free variable z of t, $z \in \mathit{dom}(\xi)$ and every free occurrence of z in t heads a subterm of the form $z a_1 \ldots a_n$ where n is the arity of $\xi(z)$ and where for some $\overline{u} \in D_{\xi(z)}$, each a_i is of the form $\eta(b_i, u_i)$. The term $\xi(t)$ is obtained from t by successively replacing subterms $z a_1 \ldots a_n$, where n is the arity of $\xi(z)$, by $o_{\xi(z)}(a_1) a_2 \ldots a_n$ if $n > 0$, and by $\xi(z)$ otherwise.

Finally, a term is in *λ-normal form* if it contains no operators other than λ and application, and if it contains no redexes.

Lemma 2 *Let ξ be an oracle assignment and let t be a term of S such that $\xi(t)$ is defined and $\xi(t) \Downarrow \kappa_v$ in S_c for some $v \in V$. Then we can effectively find $t' \in S$ in λ-normal form such that $t \leq_S t'$.*

The proof of this lemma involves a rather complicated analysis of derivations in S_c and in S. Space limitations preclude including the proof here. However, the basic idea is fairly simple. $\xi(t)$ evaluates to an atom. Consider transforming this evaluation into a "pseudo-evaluation" where all terms of the form $o_f(a_1)a_2 \ldots a_n$ are replaced by $za_1 \ldots a_n$ where $\xi(z) = f$. A subevaluation which originally reduced a term $o_f(a_1)a_2 \ldots a_n$ to an atom is replaced by a single step which simply reduces $za_1 \ldots a_n$ to itself. The result of the pseudo-evaluation is a term of the form $za_1 \ldots a_n$. We inductively can find normal forms for $a_1, \ldots, a_n$.

We can compute t' from t given just the arity information of an oracle assignment. We augment S with "pseudo-axioms" $za_1 \ldots a_n \Downarrow za_1 \ldots a_n$ for each z free in t, where n is the arity of z given by the oracle assignment, and then find a derivation e of $t \Downarrow a$ for some a. a must have the form $za_1 \ldots a_n$, and inductively repeating the procedure on the a_i gives $\overline{a'}$ such that $a_i \leq a'_i$. Hence $t \leq z\overline{a'}$. The proof of Lemma 2 guarantees that such an e exists. Strong determinism of S guarantees that any e will suffice.

With a moderate amount of extra work we can prove not only $t \leq_S t'$, but in fact $t \leq_{S'} t'$ for any S' extending S that in which evaluation is deterministic.

4 Normalization in an LF-like Subtheory of TT

Let S and S_c be as in the previous section. We assume familiarity with LF. We write $\Gamma \vdash_{LF} t \in T$ for the LF judgement that t has type T in the context Γ[3]. For t an LF term, type or kind, obtain *the erasure of* t, $\epsilon(t)$, by erasing the types of bound variables in λ-abstractions and by replacing *Type* by U_1. Extend ϵ to contexts in the obvious way. It is straightforward to show that ϵ gives an interpretation of LF in TT. In particular, if $\Gamma \vdash_{LF} T \in \mathit{Type}$ then $\epsilon(\Gamma) \vdash_{TT} \epsilon(T) \in U_1$, and if $\Gamma \vdash_{LF} t \in T$ then $\epsilon(\Gamma) \vdash_{TT} \epsilon(t) \in \epsilon(T)$. Furthermore, $=_{\beta\eta}$ in LF implies provable equality of erasures in TT. When no confusion is likely to result, we will identify an LF object with its erasure.

We first prove a lemma which gives conditions under which the conclusion of Lemma 2 can be strengthened to include the property that the λ-normal form is in fact the erasure of some well-typed LF term. The main condition is the existence of a certain class of models of an LF context. After the lemma we give several consequences of practical import.

An LF type or kind has the form

$$\Pi\, x_1 \in A_1 \,.\, \ldots \Pi\, x_n \in A_n \,.\, B$$

where B is either *Type* or atomic: of the form $P\overline{a}$ for some variable P. The *arity* of this type is n.

Let $\Gamma = [x_1 : A_1, \ldots, x_n : A_n]$ be an LF context. A *model* M of Γ is a sequence $a_1, \ldots, a_n$ of LF-like terms of S_c satisfying the following. (1) The arity of a_i is the arity of A_i. (2) $A_i[a_1, \ldots, a_{i-1}/x_1, \ldots, x_{i-1}]$ is a type in S_c and $a_i \in_{S_c} A_i[a_1, \ldots, a_{i-1}]$. (3) If A_i has arity $m \geq 0$, then, writing

$$A_i[a_1, \ldots, a_{i-1}] = \Pi\, y_1 \in B_1 \,.\, \ldots \Pi\, y_m \in B_m \,.\, C,$$

if $a_i b_1 \ldots b_m$ has a value in S_c then the value is an atom and for all j,

$$b_j \in B_j[b_1, \ldots, b_{j-1}].$$

The last condition says that the objects we put in function types only return values on arguments of the right type. This is needed in the next lemma when we prove that certain normal forms can be well-typed in

[3] here, contexts include LF signatures

LF. In S_c, every function type is fully-inhabited, in the sense of Section 2.3, by functions which have values only on the domain of the type.

Write $M \models \Gamma$ if M is a model of Γ, and write $M(t)$ for $t[\overline{a}/\overline{x}]$. If the free variables of t and T are bound by Γ, then define $M \models t \in T$ if $M(t) \in_{S_c} M(T)$.

The proof of the following is a straightforward induction on the size of t.

Lemma 3 *Let t be a term in λ-normal form. Let Γ be an LF context and let T be a term such that $\Gamma \vdash_{LF} T \in Type$. Let M be a non-empty set of models of Γ. Suppose that for all $M \in M$, $M \models t \in T$, and if T', T'' are LF types in Γ with different $\beta\eta$-long normal forms, then there exists $M \in M$ such that $M(T')$ and $M(T'')$, as types in S_c, have no LF-like members in common. Then we can effectively find a term t' such that $\Gamma \vdash_{LF} t' \in T$ and $t = \epsilon(t')$.*

The required type information for t' can be computed in a single top-down pass over t since we are given the type for t.

The following theorem directly justifies the use of TT as a logical framework in the sense of LF.

Theorem 4 *Suppose that $\Gamma \vdash_{LF} T \in Type$ and that t is a term of S such that $\Gamma \vdash_{TT} t \in T$. Then we can effectively find a t' in λ-normal form such that $\Gamma \vdash_{LF} t' \in T$ and $\Gamma \vdash_S t = t' \in T$.*

Proof (sketch). Without loss of generality we may assume that T is atomic, and t has the form required for $\xi(t)$ to be defined for some oracle assignment ξ derived from a model of Γ in the obvious way. Since T is atomic, in any model M of Γ, $M(T)$ is a base type κ_u so $M(t) \Downarrow \kappa_v$ for some $v \in u$. By Lemma 2 and Theorem 3, there is a t' in λ-normal form such that $\Gamma \vdash_{TT} t = t' \in T$. It is straightforward to show that the class M of all models of Γ satisfies the conditions of Lemma 3, so we are done. $\square$

For the statement of the next theorem, define the *simple types* over $c_1, \ldots, c_n$ to be all terms constructed from $c_1, \ldots, c_n$ and $\rightarrow$. Also, define the *base types* of a context Γ to be the variables x such that $x : A$ appears in Γ.

Theorem 5 *Let $\Gamma = [x_1 : A_1, \ldots, x_n : A_n]$ be an LF context where each A_i is either Type or is a simple type over $x_1, \ldots, x_{i-1}$. Let Ax be a type of TT (under the hypotheses Γ) representing, via propositions-as-types, a formula of higher-order logic with equality where: (1) the types of quantified variables are simple types over the base types of Γ and the (non-atom) base types of TT (for example, the type of integers); (2) the function symbols come from Γ; (3) the terms mentioned in Ax are well-typed in the sense of the simply-typed λ-calculus. Let T be a term such that $\Gamma \vdash_{LF} T \in Type$, and let t be a term of S in which r does not occur free. Suppose that*

$$\Gamma,\ r : Ax \vdash_{TT} t \in T$$

and that Γ has a model M such that for some term r of S_c, $M \models r \in Ax$. Then we can effectively find a t' in λ-normal form such that $\Gamma \vdash_{LF} t' \in T$ and $\Gamma,\ r : Ax \vdash_S t = t' \in T$.

Proof (sketch). Because of the restriction on Γ, all the LF types well-formed under Γ are simple types (composed of $\rightarrow$ and variables x such that $x \in Type$ is in Γ). Hence we can modify M, by making copies of intersecting base types if necessary, so that $\{M\}$ satisfies the conditions of Lemma 3. Then $M \models t \in T$, so we can proceed as in Theorem 4. $\square$

We can easily extend TT to include Nuprl's subset type. The type $\{\, x \in A \mid B \,\}$ has as members all terms $a \in A$ such that $B[a/x]$ has a member. The proof rules for this type are given in [6]. Define the *squash* of a type A, denoted $\downarrow(A)$, to be $\{\, x \in Unit \mid A \,\}$ where $Unit$ is a fixed type with a single member. Note that $\downarrow(A)$ has as its unique member the member of $Unit$ if A is non-empty, otherwise it is empty. The rules of Nuprl guarantee that a squashed proposition may not be used during computationally significant parts of proofs. Hence, the effect of replacing Ax with $\downarrow(Ax)$ is to remove explicit variable occurrence restrictions in Theorem 5. We can now prove the following.

Theorem 6 *Suppose* $\Gamma,\ r :\downarrow (Ax)\ \vdash_{TT}\ t \in T$ *and that all the conditions of Theorem 5 hold except the restriction on occurrences of* r *in* t. *Then we can effectively find* t' *in* λ*-normal form such that* $\Gamma,\ r :\downarrow (Ax)\ \vdash_S t = t' \in T$.

5 Applications

In this section we describe an extension of the *proofs-as-programs* paradigm to the synthesis of programs with specific intensional properties. Theorem 6 provides a theoretical basis for using the proofs-as-programs paradigm to construct proof-objects that have desired structural and behavioral properties. In this section we illustrate this with an example: a context for developing and reasoning about representations of CMOS transistor-level circuits.

Terms representing CMOS circuits (which we henceforth call "circuit terms") are defined relative to a context which provides the appropriate constructors and axioms. Primitive circuit terms are either *power*, *ground*, or are one of two kinds of transistors (n-type and p-type). Circuit terms are composed using an operator *Join*, and wires are introduced between similarly named ports by an operator *Wire*. The constructors, along with related types, are declared in the context Γ_{CMOS}, listed below.

$$T{:}U_1,\ B{:}U_1,\ \mathit{Ntran}{:}B \to B \to B \to T,\ \mathit{Ptran}{:}B \to B \to B \to T,$$
$$\mathit{Pwr}{:}B,\ \mathit{Gnd}{:}B,\ \mathit{tt}{:}T,\ \mathit{ff}{:}T,\ \mathit{Join}{:}T \to T \to T,\ \mathit{Wire}{:}(B \to T) \to T,$$

T can be thought of as the type of circuits and B as the type of ports.

Circuits terms are built in extensions to Γ_{CMOS} that contain additional declarations of port names. For example, a term t such that

$$\Gamma_{CMOS}, p_1{:}B, ..., p_n{:}B \vdash_{TT} t \in T$$

is intended to represent a circuit whose ports are named by the variables p_i. As an example, within the context Γ_{CMOS} extended by $i{:}B$ and $B{:}T$ the term

$$\mathit{Join}(\mathit{Ptran}(i, \mathit{Pwr}, o), \mathit{Ntran}(i, o, \mathit{Gnd})) \tag{1}$$

is of type T and represents a CMOS inverter.

While the context Γ_{CMOS} is sufficient for specifying and reasoning about the structure of circuits, it is inadequate for reasoning about their logical properties. We rectify this by augmenting Γ_{CMOS} with axioms for the types and constructors.

We axiomatize a simple *switch model semantics*[4], where each circuit with ports $a_1, \ldots, a_n$ is represented by a relation $R(a_1, ..., a_n)$ which is true precisely when the a_i have values that are consistent with the logic of the circuit. Under this interpretation, we may think of circuits as constraints. Composing circuits combines constraints: if R_1 and R_2 are circuits, then their join $\mathit{Join}(R_1(a_1, ..., a_n), R_2(a_1, .., a_n))$ represents a circuit which is true when the a_i satisfy both relations. Hence *Join* corresponds to conjunction. Given a function from B into T, e.g.,

$$\lambda w.\ R(a_1, ..., a_{i-1}, w, a_{i+1}, ..., a_n),$$

Wire internalizes the port w and the resulting relation should be true when there is some value for w that makes the relation true. Hence, the *Wire* operator corresponds to existential quantification over T. Transistors are axiomatized as relations that respect their switching values. For example, when the signal at the gate of an n-type transistor is high (equal to *Pwr*) than the source and the drain are connected and hence have the same value.

Our axioms for this interpretation are as follows. Here $tr(b)$ is defined as the *type* $b = \mathit{Pwr} \in B$, which encodes the proposition that b and *Pwr* are equal members of B. We will use the same notation for the analogous operation for T.

$$\begin{array}{l}
\forall b{:}B.\,(b = Pwr \in B \vee b = Gnd \in B) \wedge \neg(Pwr = Gnd \in B)\\
\forall b{:}T.\,(b = tt \in T \vee b = f\!f \in T) \wedge \neg(tt = f\!f \in T)\\
\forall g,s,d{:}B.\, tr(ntran(g,s,d)) \Leftrightarrow (tr(g) \Rightarrow (tr(s) \Leftrightarrow tr(d)))\\
\forall g,s,d{:}B.\, tr(ptran(g,s,d)) \Leftrightarrow (\neg tr(g) \Rightarrow (tr(s) \Leftrightarrow tr(d)))\\
\forall x,y{:}T.\, tr(Join(x,y)) \Leftrightarrow tr(x) \wedge tr(y)\\
\forall f{:}B \to T.\, tr(Wire(f)) \Leftrightarrow \exists b{:}B.\, tr(f(b))
\end{array}$$

We extend Γ_{CMOS} by adding the squashed conjunction of these axioms.

Within the scope of Γ_{CMOS} we may both verify that circuits terms have specific logical properties and synthesize circuit terms from specifications of their properties. To verify that an explicitly given circuit t with ports $p_1, \ldots, p_n$ satisfies a specification *Spec*, we prove

$$\Gamma_{CMOS}, p_1{:}B, \ldots, p_n{:}B \vdash t \in \{\, t{:}T \mid tr(t) \Leftrightarrow Spec \,\}$$

More interestingly, a circuit satisfying *Spec* could be *synthesized* from a proof in TT. This takes advantage of the fact that it is possible to implement TT in a way that suppresses explicit members of types. A user, or theorem proving program, deals with sequents of the form $\Gamma \vdash A$. From a complete proof of such a sequent, a term t can be synthesized, or *extracted*, such that $\Gamma \vdash t \in A$ is provable in TT. Thus a circuit t could be synthesized from a proof of

$$\Gamma_{CMOS}, p_1{:}B, \ldots, p_n{:}B \vdash \{\, t{:}T \mid tr(t) \Leftrightarrow Spec \,\}$$

Since the provability of this implies that $t \in T$ is provable in the same context, Theorem 6 gives us a t' satisfying the specification which also directly represents a circuit. Hence, our work can be seen as providing a basis for generalizing the proofs-as-programs paradigm to synthesizing proven correct circuit terms from proofs of their specifications.

As a verification example, the reader may check that the inverter circuit (1 above) is satisfied if and only if the ports i and o are assigned different values in T. That is, the following is provable in TT.

$$\Gamma_{CMOS}, i{:}T, o{:}T \vdash inv \in \{\, t{:}T \mid tr(t) \Leftrightarrow (tr(i) \Leftrightarrow \neg tr(o)) \,\}.$$

Alternatively, the circuit term may be synthesized. Any proof of

$$\Gamma_{CMOS}, i{:}T, o{:}T \vdash \{\, t{:}T \mid tr(t) \Leftrightarrow (tr(i) \Leftrightarrow \neg tr(o)) \,\}$$

constructs a circuit term that logically implements an inverter.

One advantage of working within a rich type theory like TT as opposed to the weaker LF is that we may parameterize circuit terms by arbitrary types in TT. For such parameterized terms, when the parameter types are not LF types, our normalization result no longer applies. However, for any type-correct instantiation of the parameters, the resulting term will be a member of T and Theorem 6 will again apply. This result is satisfactory as in practice those working in hardware verification generally wish ultimately to build correct "concrete" circuits (e.g., 16-bit adders) as opposed to circuit schemas (e.g., n-bit adders) themselves.

As an example of this, we indicate how we might specify an adder schema that operates on variable width numbers. Such a schema will operate on sequences of ports of the same length. Let $\{1..n\}$ denote the type $\{\, m{:}N \mid 1 \leq m \leq n \,\}$. The type of port-value sequences (or *bit-vectors*) of length n is $Bvec(n) = \{1..n\} \to B$. To specify the adder we need auxiliary functions to interpret port values as bits and sequences of port values as unsigned numbers. To accomplish the former we require a function of type[4]

$$Val = \{\, f{:}B \to \{0..1\} \mid f(Pwr) = 1 \in int \wedge f(Gnd) = 0 \in int \,\}.$$

[4] Because the context axioms are squashed they may not be used computationally; hence one may not directly define a function that computes over members of T.

To accomplish the latter, we may define by recursion on n a function $nval$ such that for any $val \in Val$, $n \in N$, and $v \in Bvec(n)$

$$nval(val, n, v) = val(v_1) + 2 * val(v_2) + \ ...\ + 2^{n-1} * val(v_n).$$

With these in hand, the specification of parameterized n-bit adders is

$$\Gamma_{CMOS} \vdash \Pi\, n \in N\,.\,\Pi\, a, b, c \in Bvec(n)\,.\,\Pi\, cin, cout \in B\,.\,\{\, t : T \mid tr(t) \ \Leftrightarrow AddSpec\,\}$$

where $AddSpec$ is the proposition

$$\begin{aligned}\forall val\colon Val.\, nval(val, n, a) + nval(val, n, b) + val(cin)\\ = \ nval(val, n, c) + 2^n * val(cout) \quad \in N.\end{aligned}$$

Suppose we have constructed a term t meeting the above specification. Theorem 6 does not directly apply since the type of t involves non-LF types. However, the theorem does apply to any application of t to specific arguments. Suppose $m > 0$. Let us extend Γ_{CMOS} with $3 * m + 2$ port names $a_1, ..., a_m$, $b_1, ..., b_m$, $c_1, ..., c_m$, cin and $cout$; these represent the ports of the two input numbers, the output number, the carry-in bit and the carry-out bit respectively. Call the resulting context Γ_{add}. Now we can write functions a, b, c of type $Bvec(m)$ such that $a(1) = a_1$, $a(2) = a_2$ etc. Since the term $t(m)(a)(b)(c)(cin)(cout)$ is in the type T under the assumptions of Γ_{add}, we can now apply the theorem as before and conclude that the term is equivalent to a circuit.

Providing a formal proof of the above specification would take us outside the scope of this paper; the reader may find details of proofs conducted in Nuprl for similar specifications in [3]. Such specifications are proved by induction and different proofs will construct terms representing different kinds of parameterized adders (e.g., ripple-carry, carry look ahead, etc.).

As with software development, the simultaneous synthesis and verification of hardware can simplify the design process. This methodology can put to particularly good advantage when circuit terms can be constructed and reasoned about in parts. [9] and [7] give examples of this kind of development methodology applied to the top down synthesis of proven correct combinational logic and sequential circuits. [3] provides examples of using type theory to synthesize recursive circuit schemas from their schematic specification (such as the n-bit adder just specified). Our results here give a method of rigorously demonstrating that this kind of development is correct within TT.

References

[1] S. F. Allen. A non-type theoretic definition of Martin-Löf's types. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 215–221. IEEE, 1987.

[2] S. F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language.* PhD thesis, Cornell University, 1987.

[3] D. A. Basin. Extracting circuits from constructive proofs. In *IFIP-IEEE International Workshop on Formal Methods in VLSI Design*, Miami, USA, January 1991.

[4] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. Elsevier Science Publishers B. V. (North-Holland), 1987.

[5] R. Constable and D. Howe. Nuprl as a general logic. In P. Odifreddi, editor, *Logic in Computer Science*, pages 77–90. Academic Press, 1990.

[6] R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[7] M. P. Fourman. Formal methods for modeling design. In *Conference on Modeling the Innovation: Communications, Automation and Information Systems*, Rome, Italy, 1990.

[8] F. Hanna, N. Daeche, and M. Longley. VERITAS+: A specification language based on type theory. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Ithaca, New York, 1989. Springer-Verlag.

[9] F. K. Hanna, M. Longley, and N. Daeche. Formal synthesis of digital systems. In *IMEC-IFIP International Workshop on: Applied Formal Methods For Correct VLSI Design*, volume 2, pages 532–548, Leuven, Belgium, 1989.

[10] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *The Second Annual Symposium on Logic in Computer Science*. IEEE, 1987.

[11] D. Howe. Computational metatheory in Nuprl. *Proc. of 9th International Conference on Automated Deduction*, pages 238–257, 1988.

[12] D. Howe. Equality in lazy computation systems. *Proc. Fourth Annual Symposium on Logic in Computer Science, IEEE*, pages 198–203, June 1989.

[13] D. Howe. On computational open-endedness in Martin-Löf's type theory. To appear in *Proc. Sixth Annual Symposium on Logic in Computer Science, IEEE*.

[14] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[15] B. Nördstrom, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *International Series of Monographs on Computer Science*. Oxford Science Publications, 1990.

[16] L. C. Paulson. Natural deduction proof as higher-order resolution. *Journal of Logic Programming*, (3):237–258, 1985.