Nuprl and its Use in Circuit Design[1]

Paul B. Jackson [2]

Department of Computer Science, Upson Hall, Cornell University, Ithaca NY 14853, USA.
`jackson@cs.cornell.edu`

Abstract

Nuprl is an interactive theorem proving system in the LCF tradition. It has a higher order logic and a very expressive type theory; the type theory includes dependent function types (Π types), dependent product types (Σ types) and set types. Nuprl also has a well developed X-Windows user interface and allows for the use of clear and concise notations, close to ones used in print. Proofs are objects which can be viewed, and serve as readable explanations of theorems. *Tactics* provide a high-level extendible toolkit for proof development, while the soundness of the system relies only a fixed set of rules.

We give an overview of the Nuprl system, focusing in particular on the advantages that the type theory brings to formal methods for circuit design. We also discuss ongoing projects in verifying floating-point circuits, verifying the correctness of hardware synthesis systems, and synthesizing circuits by exploiting the constructivity of Nuprl's logic.

Keyword Codes: F.4.1; B.6.2; I.2.3
Keywords: Mathematical Logic; Logic Design, Reliability and Testing; Deduction and Theorem Proving.

1 Introduction

Nuprl [12] is a system designed for developing general mathematical theories. It is based on a higher order logic and has a very expressive type theory, similar to Martin-Löf type theory [23]. It provides an integrated environment for many kinds of reasoning including rewriting, forward and backward chaining, and arithmetic reasoning. It has an X-windows interface for entering definitions and interactively guiding proofs of theorems. Comprehension of proofs is aided by permitting concise notations for terms and by maintaining the full structure of proofs.

---

[1]This paper appears in the proceedings of the IFIP TC10/WG10.2 International Conference on Theorem Provers in Circuit Design, which are edited by V. Stavridou, T.F. Melham, and R.T. Boute, and published by North-Holland. The conference was held in Nijmegen, The Netherlands, 22-24 June 1992. The paper accompanies a tutorial given by the author at the conference.

[2]Supported by NASA GSRP fellowship NGT-50786

Nuprl can support formal methods for digital systems design in several ways. The techniques explored by the HOL group [16] for verifying circuits can be duplicated and improved on in ways which are discussed in this paper. Current projects include producing a reusable toolkit for verifying floating-point circuits, exploiting special features of Nuprl's logic to synthesize circuits and building a formally-verified hardware synthesis system. This would allow circuit designers to exploit the benefits of formal techniques without having to become intimately familiar with the techniques themselves.

One of the goals of this paper is to persuade the reader of the usefulness of a type theory such as Nuprl's over a simpler type theory such as HOL's [15]. Nuprl's *dependent* type constructors are described in some detail, and many examples of their use are given. Type checking emerges as playing an even more important organizational role than it does in most programming languages.

Nuprl's logic is *constructive* [9], its logic lacks a couple of the axioms of classical logic in their full generality. However, this hasn't restricted our reasoning about hardware, and we discuss its possible advantages.

Nuprl borrows the idea of *tactics* from the Edinburgh LCF project. Tactics are a special class of functions which organize proof development strategies. The class of tactics is open ended; as the need arises one can design more and more sophisticated tactics. However, tactics reduce every chain of reasoning to a fixed set of rules and previously proven lemmas, so the soundness of every proof ultimately rests on just the the soundness of the rules, not the tactics.

1.1 Background

The Nuprl project at Cornell grew out of work on various program verification and synthesis systems using specialized logics [11] [8]. These specialized logics were found to be rather cumbersome and a conclusion was drawn that it would be advantageous to work in as expressive logic as possible, one which could serve as a foundation for mathematics.

Nuprl's architects were attracted to Martin Löf type theory because it had been proposed as a foundational logic and because it held a promise of being a logic in which one could automatically synthesize programs directly from proofs of their specifications.

The idea of tactics, and the higher order functional language ML to support them, come from the Edinburgh LCF project [17]. Tactics are widely used in other theorem provers such as HOL, Isabelle [24], Coq [13] and VERITAS$^+$ [18].

1.2 Terminology

We use the word *term* throughout this paper to include Nuprl's programming-language constructs, types and propositions. These terms are considered to make up Nuprl's *object language*. They are sometimes divided into two classes; *primitive terms* and *abstractions*. Primitive terms are those which are dealt with directly by Nuprl's type theory. Abstractions are terms defined in terms of other abstractions and/or primitive terms.

We write $t \in T$ to mean that the term t can have type T. We also might say that t is a well formed member of type T or simply that t is well formed if T is understood. We are using *term* here in the general sense. In Nuprl, both types and propositions have types.

1.3 Organization of Paper

Section 2 deals with Nuprl's logic and type theory in some detail. Section 3 talks about how the logic is mechanized and in particular about the various kinds of tactics. Section 4 describes the X-Windows interface. Section 5 discusses learning to use Nuprl and user support. Section 6 gives many examples of the advantages of using Nuprl's type theory for hardware verification. Section 7 lists various hardware related projects, and Section 8 summarizes the main issues discussed in the paper. Section 9 gives a few practical details about the system.

2 The Nuprl Logic

Nuprl has a higher order logic. Logical propositions are constructed from the connectives $\wedge$ (*and*), $\vee$ (*or*), $\rightarrow$ (*implies*) and $\neg$ (*not*), the quantifiers $\forall$ (*for all*) and $\exists$ (*there exists*), and atomic propositions. Examples of atomic propositions are equality over any type and order relations such as $<$ and $\leq$ on the integers and on the rationals. Nuprl has a three place equality relation, written $t_1 = t_2 \in T$. The relation is true when t_1 and t_2 are equal members of the type T.

The logic is *higher order* because one can quantify over higher order objects such as functions. Examples will be seen throughout this paper of higher order quantification.

In the rest of this section we try to give a flavor of the logic and explain Nuprl's type theory.

2.1 Sequents, Rules and Proofs

Nuprl's rules are formulated in a *sequent calculus*. A sequent in Nuprl consists of a list of 0 or more hypotheses $H_1, \ldots, H_n$ and a conclusion C. It is usually written as:

$$H_1, \ldots, H_n \vdash C.$$

The $\vdash$ symbol is often called a *turnstile*. Each hypothesis H_i is either a proposition P or a declaration $x : T$ declaring variable x to be of type T. The conclusion is a proposition. A declaration $x{:}T$ as hypothesis H_i binds free occurrences of x in hypotheses $H_{i+1} \ldots H_n$ and in C. For this reason, the order of the hypotheses is important. One can't arbitrarily permute hypotheses. Sequents are always closed; they contain no free variables. We will occasionally refer collectively to the hypotheses and conclusion of a sequent as *clauses*. A sequent is true if one can prove the conclusion C under the hypotheses $H_1, \ldots, H_n$. Sequents are used to formulate the rules of Nuprl's logic and type theory.

A rule has the general form

$$\frac{\mathcal{A}_1 \quad \ldots \quad \mathcal{A}_n}{\mathcal{C}}$$

where $\mathcal{A}_i$ and $\mathcal{C}$ are sequents and $n \geq 0$. The $\mathcal{A}_i$ are the antecedents of the rule and $\mathcal{C}$ is the consequent. Such a rule can be read top down as saying that if all the $\mathcal{A}_i$ are true, then $\mathcal{C}$ is true. The rule can also be read bottom-up as saying that in order to prove $\mathcal{C}$ is true, it is sufficient to prove that all the $\mathcal{A}_i$ are true.

Nuprl's logic is similar to Gentzen's LJ system [27]. Nearly all the logic rules, when read top-down, tell us how to introduce a logical connective or quantifier in a hypothesis or the conclusion. When read bottom-up they explain how to break down or decompose the connective. We will refer to such rules in what follows as *decomposition* rules, because it turns out that we always use these rules in a bottom-up fashion. A slightly simplified version of the rule for decomposing $\Rightarrow$ in the conclusion is:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \;\Rightarrow\; B}$$

Another rule, the hypothesis rule states that

$$\frac{}{\Gamma, A,\ \Delta \vdash A}$$

Here A and B stand for arbitrary propositions, and Γ and Δ stand for arbitrary (maybe empty) lists of hypotheses.

A proof of some proposition P in Nuprl's logic is usually constructed by starting with the sequent $\vdash\ P$. One then applies rules bottom-up, building the proof tree upwards. Since most of the rules when viewed bottom-up decompose a connective, propositions generally get simpler as one moves from the root of the tree out along the branches. Branches of a proof tree terminate with such rules as the hypothesis rule above.

In practice in Nuprl, one never invokes the rules directly. As is explained in section 3.3, one can easily automatically apply many — maybe hundreds — of rules at a time.

This style of theorem proving bears a close resemblance to the tableau method for proving theorems [26], which is commonly taught in logic courses, and which students usually find the simplest to use.

2.2 Type Theory

A *type theory* defines kinds of objects that one can talk about. Common types in programming languages are integers, strings and arrays. Type theories differ in how expressive they are, and what kinds of distinctions they make. Nuprl's type theory is more expressive than the type theories of most programming languages, or of the HOL proof system. We give several examples of uses of Nuprl's type theory in section 6.

In Nuprl, terms are considered to be intrinsically untyped, terms in and of themselves don't have a type. Type membership, $\in$, should be thought of as a two place relation on these untyped terms. If it so happens for two terms a and b, that $\in (a, b)$ is true, or using infix notation, that $a\ \in\ b$ is true, then we say that a has type b. It is usual in Nuprl for terms to have more than one type.

The main built-in types of Nuprl's type theory are as follows.

- **Integer**
  The type `Int` contains the integers. Built-in operations on integers include $+$, $-$, $\times$, $\div$ and *mod*.

- **Atom**
  The type `Atom` contains alphanumeric strings. For example `"abc"`.

- **List**
  For any type A, the type A `List` contains lists of items of type A.

- **Union**
  The *union* of any types A and B, is written as $A+B$. The idea of a union type is that an element of $A+B$ is either an element of A or an element of B and that we can always tell which. This type is sometimes called a *disjoint* union to emphasize that we can always determine which side of a union an element of a union type comes from. Another name for the union type is the sum type. The rules for formation of members of a union type are:

$$\frac{\Gamma \vdash a \ \in \ A}{\Gamma \vdash inl(a) \ \in \ A+B} \qquad \frac{\Gamma \vdash b \ \in \ B}{\Gamma \vdash inr(b) \ \in \ A+B}$$

  The rules show the tags $inl()$ and $inr()$ which we put round the elements of the types A and B to show which side of the union the elements come from.

- **Dependent Product**

  Suppose we wanted to define the disjoint union type of n types $T_1, \ T_2, \ \ldots \ , \ T_n$. We could write it as $T_1 + T_2 + \ \cdots \ + T_n$. A typical element of this type would be an element t_i from the ith type T_i tagged in some way to indicate that it came from the ith type of the union. One way to describe this tagged element is as a pair $< i, t_i >$. The type of the second component of such pairs has a type which depends on the first component of the pair. The type of the second component of the pair $< i, t_i >$ is T_i. We call the type of this pair a *dependent product* type and write it as $i\!:\!\{1 \ldots n\} \ \times \ T_i$.

  Dependent product types are also known as Σ-types and the notation in this case is $\Sigma \ i : \{1 \ldots n\}. \ T_i$. This notation reminds us that dependent product types are generalized sum types.

  The general rule for forming pairs in a dependent product

$$\frac{\Gamma \vdash a \ \in \ A \qquad \Gamma \vdash b \ \in \ B_a}{\Gamma \vdash < a, b > \ \in \ x\!:\!A \ \times \ B_x}$$

  It assumes that we have an arbitrary index type A, and that we have a family of types B_x indexed by x of type A. The rule says that if we can prove that $a \in A$ under some set of assumptions Γ, and we can also prove under the assumptions Γ that b is in the particular type B_a, then we can prove that $< a, b >$ is a well formed member of type $x\!:\!A \times B_x$.

  When it happens that the type of the second component of pairs does not depend on the first component, the dependent product type becomes equivalent to the normal product type and we use the notation $A \times B$. Sometimes for emphasis, we call this special case of the dependent product type, an *independent* product type.

- **Dependent Function**

  Suppose we want to define the product type of n types $T_1,\ T_2,\ \dots\ ,\ T_n$. We could write it as $T_1 \times T_2 \times\ \cdots\ \times T_n$. A typical element of this type would be a tuple $< t_1,\ t_2,\ \dots\ ,\ t_n >$ where $t_i \in T_i$. One way of describing this tuple is as a function which when given some index i between 1 and n, returns the term t_i. This function has a result type which depends on the argument to the function. For an argument i, the type of the result is T_i. We call the type of this function a *dependent function* type and write it as $i\!:\!\{1\dots n\} \ \rightarrow\ T_i$.

  Dependent function types are also known as Π-types and the notation in this case is $\Pi\ i:\{1\dots n\}.\ T_i$. This notation reminds us that dependent function types are good for constructing generalized products.

  We describe the general rule for forming λ terms in the dependent function type. (Lambda notation provides a convenient way of describing functions. For example, $\lambda x.x+1$ is a function for adding 1 to its argument. It has the function type `Int` $\rightarrow$ `Int`.)

  $$\frac{\Gamma\ x\!:\!A \vdash b\ \in\ B_x}{\Gamma \vdash \lambda x.b\ \in\ y\!:\!A \rightarrow B_y}$$

  The rule assumes that we have an arbitrary index type A, and that we have a family of types B_x indexed by x of type A. The rule says that if we can prove that $b \in B_x$ for any value of x of type A, then we can prove that the function $\lambda x.b$ is a well formed member of the dependent function type $y\!:\!A \rightarrow B_y$.

  When it happens that the result type of a lambda function does not depend on the argument to the function, the dependent function type becomes equivalent to the normal function type and we use the notation $A \rightarrow B$. Sometimes for emphasis, we call this special case of the dependent function type, an *independent* function type.

- **Set**
  The *set* type is similar to the dependent product type. We describe it by the rule:

  $$\frac{\Gamma \vdash a\ \in\ A \quad\ \Gamma \vdash P_a}{\Gamma \vdash a\ \in\ \{x\!:\!A|P_x\}}$$

  Here P_x is some arbitrary predicate on the variable x.

- **Recursive**
  The Nuprl type theory allows one to construct a variety of recursive types, such as trees, for example.

- **Type Universe**

  There is an infinite ascending hierarchy $U_1,\ U_2,\ U_3,\ \dots$ of *type universes*. All types which don't mention a universe term are elements of the first universe U_1.

Type universes in Nuprl can, for many purposes, be treated just like other types, and so can appear in type expressions. The second universe U_2 contains all those types which are elements of U_1 and also those types which mention U_1. The higher universe levels are defined similarly.

- **Proposition Universe**

  Proposition universes $Prop_1, Prop_2, Prop_3, \ldots$ are very similar to type universes except their elements are propositions.

It should be noted that since type universes and proposition universes *are* types, types and propositions can be considered to be data. They can be put into pairs, and functions can be defined which take propositions and types as arguments, and return propositions and types as results.

2.3 Type Checking

Most theorem provers in use type-check terms automatically. The rules which govern the type-checking are kept separate from the rules of the logic and sometimes are not even mentioned explicitly. This automatic type-checking occurs at a few well-defined points in the theorem-proving process, often only at the start, and otherwise one always assumes that terms are well formed and have the correct types.

The type theory of Nuprl is expressive enough to admit formally undecidable type-checking problems. (To say that a problem is undecidable is to say that there is no way in every situation to find the problem's solution in a finite amount of time.) Fortunately, undecidable checking problems rarely come up in practice in Nuprl. However, because of this theoretical undecidability, all of Nuprl's type checking rules are explicitly-stated and logic rules include extra well-formedness antecedents.

Consider the Nuprl rule for decomposition of $\forall$ in the conclusion:

$$\frac{\Gamma,\ x\!:\!A \vdash\ B \qquad \Gamma \vdash A\ \in\ U_i}{\Gamma \vdash \forall x\!:\!A.\ B}$$

The left antecedent $\Gamma,\ x : A \vdash\ B$ is as one might expect. The right antecedent $\Gamma \vdash A\ \in\ U_i$ is a well-formedness antecedent. It demands that one prove A to be a well formed type under the assumptions in Γ.

Nuprl's rules are arranged so that whenever a proof of some proposition P is completed, a proof of the well formedness of P will have been completed as well.

Nuprl has a tactic for proving well-formedness antecedents automatically. This tactic is described in section 3.3.6.

2.4 Encoding Higher Order Logic in Type Theory

Up to this point, we have made a clear distinction between Nuprl's higher-order logic, and Nuprl's type theory. This distinction is also clear when one uses the Nuprl system.

The truth of the matter is that none of the rules of Nuprl's logic deal with propositions. They are all type theoretic. All the higher-order logic term constructors are abstractions, non primitive definitions.

Martin-Löf's type theory, on which Nuprl's is based, was specially designed to encode a higher-order logic. The encoding is as follows:

$$\begin{array}{rcll} \bot & =_{def} & Void & \\ A \wedge B & =_{def} & A \times B & \\ A \vee B & =_{def} & A + B & \\ A \Rightarrow B & =_{def} & A \rightarrow B & \\ \forall x\!:\!A.\ B_x & =_{def} & x\!:\!A \rightarrow B_x & (\Pi\ x\!:\!A.\ B_x) \\ \exists x\!:\!A.\ B_x & =_{def} & x\!:\!A \times B_x & (\Sigma\ x\!:\!A.\ B_x) \\ Prop_i & =_{def} & U_i & \end{array}$$

$Void$ is the empty type. $\bot$ is falsity. $\neg A$ is defined as $A \Rightarrow \bot$. As one can see, the encoding is very direct.

To seek the truth of some logical proposition P using type theory, one translates P into an equivalent type T and asks whether there exists a term of type T. One then uses the rules of type theory to try to answer this question. Some view the fact that one can do this at all as purely accidental. Others point out that mathematicians of the *Intuitionistic* or *Constructive* school [9] have been developing related ideas since the early years of this century. This equivalence between logic and type theory is known as the Curry-Howard isomorphism.

2.5 Proofs as Programs

Interest in studying the Curry-Howard isomorphism stems from the fact that from the proof of the existence of a term t of type T, one can actually construct t. This term is called the *extract* of the proof. Often t or part of t will be a function, and one can view the proposition P as a specification of the behavior of t. Thus type theory offers an intriguing possibility for the automatic production of correct programs directly from their specifications.

The chief problem with producing programs this way is that proof procedures have to be sensitive to the efficiencies of the computations implicit in them. If conventional proof techniques are used, extremely inefficient programs can result.

2.6 Constructivity

Nuprl's higher order logic is *constructive* in contrast with, for example, HOL's, which is *classical*. The difference is that in Nuprl's logic the axiom of the excluded middle,

$$\vdash \forall P\!:\!U_i.P \vee \neg P$$

and the axiom of choice (an element can be chosen from each of an infinite collection of non-empty sets) are not true in their full generality. A classical higher order logic can be encoded inside Nuprl's constructive logic. However, we haven't yet found this necessary for our work in hardware verification. All the instances of these axioms which have come up and which we anticipate coming up are true in Nuprl's constructive logic.

2.7 Semantics

One can go a long way with Nuprl's logic with a reasonably straightforward idea of what the types mean and how the rules work. However, it is not easy to make these intuitive ideas precise. Several semantics have been proposed for Nuprl's type theory. The most fully elaborated is Allen's set-theoretic model [2]. Nearly all of the rules have been shown to be correct with respect to this model. However, this model is rather complex and it is a non-trivial task to verify the soundness of new extensions to the logic.

2.8 Discussion

Nuprl's type theory does have a few problems, aside from the complex semantics mentioned in the previous section. There are some technical difficulties with type membership propositions that prevent one from always being able to summarize certain patterns of inference as lemmas rather than large tactics. For example induction schemes over arbitrary well-founded orders. It also would be nice to find a systematic way to avoid the duplication of well-formedness subgoals which frequently occurs. We partially deal with this problem now by reusing well-formedness proofs so proofs become directed acyclic graphs rather than trees.

The particular Nuprl rules and types were chosen for pragmatic reasons. An easy to use and efficient logic was desired. The designers didn't strive to define as Spartan a logic as possible as happens when logicians design logic. For example the integers with the basic arithmetic operations are built in, rather than being derived from Peano's axioms or the equivalent. For this reason and because we need to have explicit rules to check types, Nuprl's logic has a relatively large number of rules. (On the order of one hundred and twenty.)

3 Mechanization of Logic

3.1 Abstractions

Abstractions are Nuprl's version of definitions.

We call the process of expanding a definition *unfolding*. The reverse process is *folding*. For example, the abstraction for negation in Nuprl's logic reads:

```
¬ P == P => False
```

Abstractions can be made for terms which have binding structure. For example, existential quantification is defined in terms of the dependent product type:

```
∃ x:A. B[x] == x:A × B[x]
```

3.2 Proofs

One major difference between Nuprl and the other LCF style theorem-provers is that in Nuprl the tree structure of proofs is maintained as proofs are developed. A node of a proof tree contains the sequent at that stage of the proof as well as a record of the tactic invoked on that node to expand the proof tree. In contrast, in the other systems, only the fringe of the proof tree is maintained; to make a transcript of the tactics used in a proof legible, one must carefully structure and annotate the transcript by hand. Nuprl's proof trees serve as readable explanations of a proof, and simplify backtracking and trying alternative proof development strategies.

Note that the proof tree that a user sees is *not* a proof tree at the level of the basic Nuprl rules. Rather, the children of a node are the unproven leaves of the proof tree left over after the run of a tactic at that node.

We are investigating ways of partially expanding sections of a proof, so that if one doesn't understand why a tactic has behaved some way, one can discover the sequence of slightly more primitive tactics that the tactic invoked in a simple way. Currently, we can expand part of a proof down to the primitive rule level, but this expansion is usually far too much.

Proof trees can be large data-structures. When we save a proof, we only store its main goal and a transcript of the tactics required to build it. When a user asks to view a proof, it is regenerated from the transcript.

3.3 Tactics

Tactics make up a very high-level language in which to write decision procedures and proof development heuristics. This language consists of a set of higher order functions. Nuprl has two kinds of tactics; *refinement tactics* and *transformation tactics*. The difference is that transformation tactics are usually run on internal nodes of proof trees, and refinement tactics are run on incomplete leaf nodes of proof trees. (By incomplete, we mean that a rule with no antecedents has not been applied to that node.) Unless otherwise specified, we will only be concerned with refinement tactics from here on.

A Tactic can be thought of conceptually as a function which takes an incomplete leaf node as an argument, grows the proof tree by one or more rules, and then returns as a result a list of the incomplete leaf nodes which are at the fringe of the growth achieved by the tactic. We often call the node of the proof tree that a tactic is run on the current *goal*, and the fringe of nodes that the tactic returns the *subgoals* generated by the tactic.

Tactics are combined using *tacticals*. For example, if `T1` and `T2` are tactics, then `T1 THEN T2` is also a tactic which runs `T1` and then runs `T2` on each of the subgoals produced by T1. `T1 ORELSE T2` runs `T1` and if `T1` fails, it tries `T2` instead.

Tactics are written in ML, a functional programming language specifically developed for tactic writing [17]. The tactic language is an elegant demonstration of the virtues of higher-order functional programming.

All tactics in the current collection attach *labels* to the subgoals they produce. A label describes the kind of a subgoal or what tactic produced it. Labels help users sort out where subgoals come from. There are also tacticals which discriminate on labels. For instance we nearly always want to run our type checking tactic on well formedness goals produced by other tactics. Before we had these tacticals, we made extensive use of a tactical called `THENL`. `T THENL [T1;...;Tn]` applies `T` and then `T1 ... Tn` to the `n` subgoals of T. It requires the user to figure out the order in which subgoals of `T` come out, an unrewarding and tedious task. Labels increase the robustness of tactics by making their behavior easier to understand and hence more predictable.

We also are exploring analogous ways to index into the hypothesis list, such that tactic transcripts of proofs can be replayed successfully, even when extra hypotheses are added or the order of hypotheses is altered. Such changes, for example, are needed when one gets half way through a proof of some theorem, and suddenly realizes that one needs an extra assumption to be stated in the theorem for the theorem to be true.

One of the chief virtues of tactics is that they provide a systematic way of interacting with Nuprl at a variety of levels of abstraction; Some tactics invoke maybe one or two primitive rules. Others can result in the invocation of hundreds or thousands of rules. They partition automatic theorem proving strategies into well-defined, predictable modules, that one can use selectively.

The main classes of tactics in Nuprl are as follows:

3.3.1 Decomposition

The simplest of these tactics invoke single decomposition rules for type and logic constructors. Others invoke well formedness checking rules. We try to write tactics which group rules with similar functions, to reduce the number of tactic names the user need remember. Another example of tactics in this class are instantiation tactics for universally quantified hypotheses and lemmas.

3.3.2 Induction

There are tactics for induction over the integer and lists types as well as induction over various subtypes of the integers. We are working on providing induction lemmas for more complicated induction schemes, such as induction over well founded orders.

3.3.3 Forward and Backward Chaining

These tactics work with hypotheses and lemmas constructed mainly out of $\forall$ quantifiers, $\wedge$'s and $\Rightarrow$'s. Such formulae behave as derived rules of inference, often at a high level. Backward chaining is similar to resolution in Prolog. The difference in Nuprl is that matching rather than unification is used, and one has much more control over the search procedure. The basic step of backward chaining involves matching the conclusion proposition against the consequent of the derived rule. This results in the production of a series of subgoals — one for each antecedent of the derived rule — which one in turn tries to backchain with.

A forward chaining step involves the matching of hypothesis propositions against the antecedents of derived rules. It results in the consequent of the rule being added as a new hypothesis.

3.3.4 Decision Procedures

Nuprl has decision procedure tactics for equality and arithmetic reasoning. These tactics act as interfaces between the user and the basic rules which invoke the procedures coded in Lisp.

The arithmetic decision procedure tactic `Arith` works over the integers and subsets of the integers. It handles equality reasoning and some simple inequality reasoning.

`Arith` cannot handle arbitrary integer linear programming problems. These come up frequently when one type-checks types parameterized by integers. For this reason, we are implementing the *Sup-Inf* method for linear programming problems over the integers and the rationals [10]. We hope to verify this implementation using our work on reflection. See section 3.4 for details.

3.3.5 Rewriting

Term rewriting is a very useful technique for theorem proving. Some systems base all their reasoning on rewriting [14]

Nuprl has a term rewriting package using *conversions* [25]. Conversions form a very high-level modular language for building rewriting strategies. The style of this language is similar to that of the tactic language. Simple conversions are combined into more sophisticated ones using *conversionals*. Conversions provide a lot of control over the rewrite process. They can be used both to carry out single rewrite steps on particular subterms and to simplify expressions into normal forms using complete sets of rewrite rules.

The package supports rewriting with respect to user-defined equivalences and order relations as well as the $\Leftrightarrow$ (*if and only if*) and $=$ relations of Nuprl's logic. An example of an order relation is $\leq$ over the integers. Order relations are common when one is dealing with process algebras or program refinement logics. Also, when $\Rightarrow$ is treated as an order relation on propositions, rewriting turns out to be a generalized form of forward and backward chaining. Conversions can be derived from lemmas or hypotheses. There are also conversions for folding and unfolding abstractions, and for evaluating functions.

Conditional conversions are supported. The user has control over how the system decides when a conditional conversion applies.

Conversions automate the generation of justifications for rewrites, proofs that the rewrites are valid. In general Nuprl's type theory doesn't allow one to replace a subterm by an equal subterm without such a proof. However, proofs don't have to be generated when folding and unfolding abstractions, and when evaluating functions.

3.3.6 Type Checking

As mentioned previously, the Nuprl logic has explicit type-checking rules, and type-checking is performed by a tactic.

Nuprl's type checking tactic is always run after other tactics which generate well-formedness subgoals. It proves these subgoals automatically so the user never sees them in the normal run of events.

The type checking tactic checks a term's well-formedness working from the top down (the outermost term constructor inwards). Sometimes it needs to guess types to supply as arguments to rules, and a bottom-up type inference program in ML takes care of this. We have found this scheme to work well in practice.

The user supplies one or more typing lemmas for most abstractions. The type checking tactic and type inference routines make use of these lemmas. These lemmas make clear the intended use of abstractions. Often they can't be proved automatically; a little user guidance is required. It is not unusual to have to use some kind of induction to prove a well-formedness lemma.

The type checking tactic often needs to recognize when one type is a subtype of another type, and we provide systematic ways for the user to indicate those inclusion relationships on which Nuprl should act.

3.3.7 Transformation

The main transformation tactics are for reasoning by analogy. They take a transcript of the tactics from some branch of a proof tree, and apply those same tactics to some other branch. The labelling techniques discussed above are improving the robustness of these tactics. We also use a transformation tactic for pretty-printing proof trees.

3.4 Reflection

One drawback of the tactic style of reasoning is that it is often very inefficient for tactics to have to construct proof justifications using primitive inference rules and preproven lemmas. A user is going to be understandably irritated when an inference procedure takes minutes or hours when it would take seconds if hard coded.

One ad-hoc partial solution used in the current version of Nuprl has been to code a few critical inference procedures as lisp or ML programs rather than as tactics. Our confidence in the soundness of inferences in Nuprl, is based not only on our confidence in rules which work by matching and substitution but also on our confidence in these inference procedures. Another ad-hoc solution we use is to provide two modes of operation for some tactics, one fast and unsafe, another slow and safe. A user interactively develops proofs in the fast and unsafe mode, and then replays the proofs overnight in the slow and safe mode. For this solution to work, one has to ensure that both versions of a tactic always have the same behavior. This solution works well with rewriting, where the process of calculating the effect of a rewrite is separate from, and often much faster than, the process of proving the rewrite valid.

To improve this situation, we are actively pursuing work in an area called *Reflection*. We are building as a Nuprl theory a complete model of the Nuprl logic [19] [3]. When the model is complete, we will verify the correctness of inference procedures, written in Nuprl's object language, which manipulate the data-structures of the model. These verified inference procedures will then be invoked using a specially designed reflection rule. Reflection promises the speed of hard coded inference procedures, combined with the reliability of the tactic style of reasoning, without the difficulties of maintaining two compatible versions of tactics.

3.5 Library

Nuprl has a library mechanism for organizing theories. A *library* is a linear list of various kinds of objects. An object in a library can only depend on earlier objects in the same library. Most objects in the library are what we call *text/term* objects. These objects are edited using the *text/term editor* as described in section 4.5. Objects in this class include *rule* objects for holding statements of Nuprl's rules, and *abstraction* objects holding abstraction definitions. Theorem objects are more complex. They hold statements of theorems and the proof trees justifying the theorems. Work is under way at Cornell to move to a more sophisticated library structure based on a partial rather than a linear ordering of objects.

4 The User Interface

4.1 Introduction

Nuprl has an X-Windows interface. The primary windows are the *command window* and the *library display* window. Instances of two other kinds of windows are created from the command window as and when needed. A proof editor is used for developing proofs of theorems and viewing the tree structure of proofs, and a *text/term editor* window is used for entering and viewing library objects such as abstractions, rule definitions and display forms. Text/term editor windows are also invoked from within the proof editor in order to enter theorem statements and tactics. Examples of most of the kinds of windows are given in the following sections.

The mouse can be used to both move between windows and to move within a window. Within a proof editor window the mouse can be used to walk the proof tree and invoke term/text editor windows.

4.2 Term Display

In Nuprl the user can control the form in which a term is displayed. The display-form of a term is independent of its logical structure and so users are free to customize the display of terms to their taste. To this end, the library includes *display form* objects for each and every kind of term constructor. Display form objects say what text to use to represent a term and where to insert the text representing the subterms. Examples are given in figure 1.

A *precedence* can also be specified for each term constructor. The precedence information controls the adding of parentheses to make the display forms of terms unambiguous. A pretty-printing routine adds line breaks and indentation to terms to further improve readability. Display forms for Nuprl's X-Windows interface use a single size ASCII font extended with assorted graphics characters. Display forms for library and proof listings can use display forms defined in LaTeX. We plan to extend the X-Windows interface in order to cope with LaTeX display forms in the near future.

```
Library (File:  ./lib/test/basics1.lisp) @ turing2
*D true_df          True ;;
*A true             True == 0 ∈ Int ;;
*D false_df         False ;;
*A false            False == Void ;;
*D and_df           <prop> ∧ <prop> ;;
*A and              P ∧ Q == P × Q ;;
*D or_df            <prop> ∨ <prop> ;;
*A or               P ∨ Q == P + Q ;;
*D implies_df       <prop> => <prop> ;;
*A implies          P => Q == P → Q ;;
*D rev_implies_df   <prop> <= <prop> ;;
*A rev_implies      P <= Q == Q => P ;;
*D not_df           ¬<prop> ;;
*A not              ¬A == A => False ;;
*D iff_df           <prop> <=> <prop> ;;
*A iff              P <=> Q == (P => Q) ∧ (P <= Q) ;;
*D exists_df        ∃<var>:<type>.  <prop> ;;
*A exists           ∃x:A. B[x] == x:A × B[x] ;;
*D all_df           ∀<var>:<type>.  <prop> ;;
*A all              ∀x:A. B[x] == x:A → B[x] ;;
*C THEOREMS         A couple of theorems from propositional and predicate calcul
*T prop1            ∀A:Prop1.  ∀B:Prop1.  ¬A ∨ ¬B => ¬(A ∧ B)
*T pred1            ∀T:U1.  ∀A:T → Prop1.  ∀B:Prop1.  (∀x:T. A[x] => B) <=> (∃
```

Figure 1: The Library Display Window

4.3 Library Display

The library display shows a section of the library with a one-line summary for each object. An example is shown in figure 1. Each line shows from left to right a status (`*` means the object is complete), a kind (`A` is for abstraction, `C` is for comment, `D` is for display form and `T` is for theorem), a name and the beginning of a description. One can view an object by name to see its full description.

4.4 Command Window

The command window is for terminal-style interaction with Nuprl. Command line editing features are provided. The window is also used for error messages such as a notice that an attempt to use some tactic has failed. The chief modes of operation of the command window are the *command* mode for managing the library and calling up library objects, the *ML* mode for exploring Nuprl's data-structures and developing tactics, and the *Eval* mode for evaluating programs written in Nuprl's object language.

4.5 Text/Term Editor

Text/term objects are conceptually like small text files, except that some characters are replaced by Nuprl terms. Terms are usually not entered as text strings which are parsed, but rather using editor for tree structures. When entering terms, one calls up display form templates for term constructors by name. Several examples of these display form

```
EDIT THM pred1 @ turing2
* top
>> ∀T:U1.  ∀A:T → Prop1.  ∀B:Prop1.  (∀x:T. A[x] => B) <=> (∃x:T. A[x]) => B

BY RepeatM (D 0) THENA Auto

1* 1.  T: U1
   2.  A: T → Prop1
   3.  B: Prop1
   4.  ∀x:T. A[x] => B
   5.  ∃x:T. A[x]
   >> B

2* 1.  T: U1
   2.  A: T → Prop1
   3.  B: Prop1
   4.  (∃x:T. A[x]) => B
   5.  x:  T
   6.  A[x]
   >> B
```

Figure 2: The Proof Editor Window

templates can be seen in figure 1. The lines with display form templates start with `*D`. When the subterm slot of a term not instantiated, the display form generator fills the slot position with a label defined in the display form object for the term. The `<prop>` and `<var>` are examples of these labels. They serve to remind one what belongs in each slot.

4.6 Proof Editor

The proof editor is used when developing and viewing proofs. An example of the proof editor window is shown in fig 2. This is the root node of the proof of the theorem `pred1` in the library display. `top` indicates the nodes position in the proof tree. The `*` by `top` indicates that the proof tree from this point down to the its leaves is complete. On the next line is the sequent associated with this node — the statement of the theorem being proved. The `>>` is Nuprl's version of the sequent turnstile. `BY RepeatM (D 0) THENA Auto` was the tactic typed in for the first step of the proof. `D 0` means decompose the conclusion term. `RepeatM (D 0)` means keep trying to decompose the conclusion of the main subgoals. (Don't try decomposing auxiliary type checking subgoals.) `THENA Auto` is a directive for the type checking tactic `Auto` to run on these auxiliary subgoals. Main and auxiliary subgoals are more reliably distinguished using labels as discussed in section 3.3, rather than using the form of the propositions or types in subgoals. `Auto` takes care of all the well formedness checking so that we end up with just two subgoals. The numbers in each subgoal enumerate the hypotheses of the subgoal sequents.

5 Learning to Use Nuprl/ User Support

In order to begin using Nuprl, it helps if one is familiar with predicate logic. That is, one should have at least a passing acquaintance with propositions, quantifiers, rules, sequents and proofs. It also helps to be familiar with the ideas of types found in programming languages such as Pascal, Ada, and in particular, SML.

The first things one must learn are the basics of getting around the system: how to get Nuprl started and load a basic library, and how to use the different editors. Proving theorems from propositional and predicate calculus is probably the first thing to try. In order to do this, one need only use a couple of variants on the decomposition tactic, and the type checking tactic.

Ideally, from this stage on, one should work through a graded series of exercises which introduce the user to the various kinds of tactics, the possibilities for combining tactics to make new, more powerful tactics, and the different types in Nuprl's type theory. One should also start looking at different theories built in Nuprl as case studies.

Such a tutorial does not, however, presently exist. There are a number of libraries that one can browse, but none are that polished. Documentation exists for parts of the system, but it is not comprehensive, and some is outdated. We are working on improving this situation.

Nuprl has several key features which make it easy to learn; a sophisticated X-Windows interface, clear and concise notation for terms, proofs which are readable, and the structured set of proof development strategies encoded in the tactics.

Nuprl is a large system, but very few users need to be familiar with its Lisp implementation; The ML language provides a very effective intermediate level of abstraction for interaction with the system. All the major data structures are accessible from ML.

Type theory does have its subtleties, but by and large, users can work with fairly simple intuitive ideas about what type theory is.

Nuprl has been used at a few sites other than Cornell. Most notably a course on theorem proving, taught to undergraduates at the University of Leeds, used Nuprl for some practicals.

6 Examples of Types

Every example in this section shows the advantages of a type theory with dependent product and subset types. Although none of the examples here explicitly mention the use of dependent function types, we are finding roles for them as generalized product types and for creating record types.

6.1 Subset Types

The set type is useful for constructing subsets of existing types. The commonest subsets we use are subsets of the integers. For example:

$$\mathtt{N} \;=_{def}\; \{i : \mathtt{Int} | 0 \leq i\}$$

$$\begin{array}{lll}\texttt{N}^{+} & =_{def} & \{i:\texttt{Int}|0<i\} \\ \texttt{Bool} & =_{def} & \{i:\texttt{Int}|i=0 \vee i=1\}\end{array}$$

6.2 Parameterized Types

The set type is also useful for constructing parameterized types. For example:

$$\{i \ldots\ j\} =_{def} \{k:\texttt{Int}|i \le k \le j\}$$

is a type parameterized by the integers i and j.

6.3 Total Functions

All functions in the Nuprl type $A \rightarrow B$ must be *total* on the domain A. In other words, for any argument in A they must terminate and return a value in B. Consider modelling an array of elements of some type T with lower bound min and upper bound max as a function f. In Nuprl we could give f the type

$$\{min \ldots\ max\} \rightarrow T$$

If we didn't have subset types, we might have to give f type

$$\texttt{Int} \rightarrow T$$

and we would have to define f to return values in type T for array indices below min and above max. To avoid the risk of inadvertently applying such an array to an out of bound index, we would have to have explicit bounds checking preconditions in all our theorems which use arrays.

This problem comes up in HOL which has total functions but no subset types.

6.4 Polymorphic Types

Consider the function π_1, the function for selecting the first element of a pair. In Nuprl we would write for it the typing lemma[3]

$$\vdash \forall A, B{:}U_i.\ \ \forall p{:}(A \times B).\ \ \pi_1(p)\ \in\ A$$

π_1 is a *polymorphic function* because it operates over pairs of any product type in a uniform way. General purpose utility functions in theorem proving are often polymorphic.

[3]Here is an example of implicit quantification over all universe levels i. This is a new feature of the Nuprl system.

6.5 Rationals

A typical rational number is commonly represented as a fraction of two integers, n/d, where $d \neq 0$.

In type theory, one can model rationals as pairs $< n, d >$, and one can define the type of rationals as:

$$Q =_{def} \texttt{Int} \times \{i\!:\!\texttt{Int} | i \neq 0\}$$

If one had to use to the type $\texttt{Int}\times\texttt{Int}$, one would have to carry around explicitly extra predicates stating that the denominator of every rational is non zero.

Equality of rationals is not the equality of pairs. The pairs $< 1, 1 >$ and $< 2, 2 >$ are not equal as pairs, but are equal as rationals.

We can define an equality over the rationals, written $=_q$ as:

$$x =_q y \quad =_{def} \quad x_n \ * \ y_d \ = \ y_n \ * \ x_d \ \in \ \texttt{Int}$$

Here x and y are pairs, x_n is the first component of the pair x and x_d is the second component.

Nuprl's term rewriting tactics supports rewriting with respect to such user defined equalities. We have to deal with rational arithmetic in our work on verifying floating-point circuits which is discussed more in section 7.1.

6.6 Bit-vector Definitions

When modelling bit-vectors in a type theory, one might start with a single type, call it `BVec`, for bit-vectors of any length.

For the purposes of this discussion, let us assume that one wants to use a two value logic and that one makes the definition

$$\texttt{BVec} =_{def} \texttt{N} \rightarrow \texttt{Bool},$$

We model a bit-vector as a function from the naturals to the booleans. This is a model which has been used in the HOL system.

The value of the ith bit of a bit-vector v of type `BVec` is obtained by applying the function v to i.

`BVec` is only a partial model for bit-vectors. A bit-vector $v \in$ `BVec` is defined for all natural numbers, whereas in practice bit-vector have fixed finite lengths. This length has to be implicitly understood in theorems which use this type. One can add to all one's theorems explicit bounds checking predicates for every index to a bit-vector, but this is clumsy and unnecessarily complicates theorems and their proofs.

With the ability to parameterize types, one can try the definition

$$\texttt{BVec}(n) =_{def} \{0 \ldots\ n-1\} \rightarrow \texttt{Bool},$$

This type transfers the responsibility of array bounds checking to the type checker. As we discuss later, there are decision procedures for the inequalities which result from bounds checking, so type checking can be done automatically. With this type, theorems need not be cluttered with explicit bounds checking predicates, and one would normally never even see bounds checking subgoals.

This type still doesn't quite capture one's intuitive notion of what a bit-vector is. For example, the following theorem is provable:

$$\vdash \forall m, n : N.\ \forall v : \texttt{BVec}(n+m).\ \ v\ \in\ \texttt{BVec}(n)$$

This is so because $v(i)$ is defined for $0 \leq i \leq n+m-1$, hence is also defined for $0 \leq i \leq n-1$, and so is in $\texttt{BVec}(n)$.

This is disconcerting because one likes to think of bit-vectors having definite lengths. One also likes to thinks of a bit-vector's length being an intrinsic part of its definition. There is no way of looking at an object of type $\texttt{BVec}(n)$ and telling its length. One might define functions on bit-vectors which need their length and one doesn't want to have to pass the bit-vector length as an extra argument. (For example consider a function for appending two bit-vectors.)

A solution is to use a dependent product type for `BVec`:

$$\texttt{BVec} =_{def} n{:}\texttt{N} \times (\{0 \ldots\ n-1\} \rightarrow \texttt{Bool}).$$

An element $< n, f >$ of this `BVec` type has the properties we expect. It's length is n and we can only find bit values by applying f to some index i for i which are in range, i which satisfy $0 \leq i \leq n-1$. However this type still is not one wants. It contains bit-vectors of all lengths, not just one specific length. We use a subset type to construct a type of bit-vectors parameterized by the length n:

$$\texttt{BVec}(n) =_{def} \{v{:}\texttt{BVec} | n = |v| \in \texttt{N}\}$$

Here the length function $|.|$ on bit-vectors is defined as the function which selects the first component of a pair.

An example of the use of $\texttt{BVec}(n)$ is the following typing lemma for a relation modelling the implementation of a parametrized `n`×`n` bit multiplier with inputs `u` and `v` and output `w`.

$$\vdash \forall n{:}\texttt{N}.\ \forall u, v{:}\texttt{BVec}(n).\ \forall w{:}\texttt{BVec}(2*n).\ \ \texttt{IntMultImp}(n)(u,v,w)\ \in\ Prop_1$$

With such a typing lemma, an instance of $\texttt{IntMultImp}(n)(u,v,w)$ will type check properly if and only if the bit-vectors u, v and w have the correct lengths. Type checking takes on the role of design rule checking, checking that a circuit has been put together properly.

6.7 Circuit Specifications as Types, and Parameterized Circuits

Assume we have relation `Imp` which models the implementation of a circuit with some set of inputs `i` of type `I` and outputs `o` of type `O`. Say `Imp` has type `(I × O) -> Prop1`. Assume also we have a relation `Spec` also of type `(I × O) -> Prop1` which specifies the intended behavior of the circuit. The statement of correctness for the circuit might have form:

$\vdash$ ∀ i:I. ∀ o:O. Imp(i,o) => Spec(i,o)

This statement is only a partial statement of correctness. It assumes that `Imp` has been constructed properly. The theorem is trivially true if `Imp(i,o)` is false for all `i` and `o`. To remedy this problem, we might want to include in the statement of correctness a condition about the circuit having defined outputs for all inputs. Such an extra well formedness kind of proof obligation is not just useful for checking that the design of the circuit is well formed, but also good for checking that the relational model of the circuit has been constructed properly. We write the theorem of correctness abstractly using a relation `ImpMeetsSpec` on the `Imp` and `Spec` relations:

$\vdash$ ImpMeetsSpec(I,O,Imp,Spec)

We can define the type of all circuit relations which implement `Spec` as:

SpecType(I,O,Spec) $=_{def}$
  {imp:(I × O) -> Prop1|ImpMeetsSpec(I,O,imp,Spec)}

and we can rephrase the theorem of correctness of the implementation `Imp` as a type checking problem:

$\vdash$ Imp ∈ SpecType(I,O,Spec)

This reformulation comes into its own when we are verifying a hierarchically designed circuit in which we wish to parameterize modules by submodules. Assume that we are designing a circuit with input type `I'` and output type `O'` which must satisfy some spec `Spec'`. That is we need an implementation `Imp'` which satisfies the theorem:

$\vdash$ ImpMeetsSpec(I',O',Imp',Spec')

or equivalently:

$\vdash$ Imp' ∈ SpecType(I',O',Spec')

We notice that the design of `Imp'` requires a submodule which meets the `Spec` we talked about first. We want to write `Imp'` as a parameterized circuit:

Imp' $=_{def}$ ParImp'(X)

where `X` is some arbitrary implementation which satisfies `Spec`. For the time being, we don't want to say anything more about this submodule. We can prove the correctness of `ParImp'` by proving the following lemma:

$\vdash$ ∀ imp:SpecType(I,O,Spec). ParImp'(imp) ∈ SpecType(I',O',Spec')

When we do get round to designing a particular implementation `Imp` for `Spec`, the statement of correctness for the instantiated circuit `Imp' = ParImp'(Imp)` is simply:

$\vdash$ ParImp'(Imp) ∈ SpecType(I',O',Spec')

We have shown in this example how the use of subset types to create such sharply defined types such as `SpecType` allows a compact and precise presentation of proofs of circuit correctness. Without subset types, the same presentation would not be nearly as concise.

6.8 Abstract Data Types

An *abstract* data type is specified by giving the type a name, and describing the valid set of functions over it. All that is said about the functions is that they satisfy some predicates. Abstract data types are useful in verification tasks. They allow one to focus on important properties and ignore irrelevant detail. Consider for example *monoids*. A monoid is a type M with an identity $e \in M$ and a binary function $o \in (M \times M) \rightarrow M$ where:

$$\begin{array}{l} \forall x, y, z : M.\ \ (x\ o\ y)\ o\ z \ = \ x\ o\ (y\ o\ z) \\ \forall x : M.\ \ x\ o\ e \ = \ x \ = \ e\ o\ x \end{array}$$

Monoids are common algebraic objects. For instance consider the integers with 0 for e and $+$ for o, or boolean logic with T for e and $\wedge$ for o. It's worth recognizing such algebraic structures when theorem proving, because then one can prove general theorems about them and use the theorems again and again in special cases.

In Nuprl's type theory one could define a monoid as the type:

$$\begin{array}{lll} \mathcal{M} & =_{def} \{< M, e, o > : & (M' : U_1 \ \times \ M' \ \times \ ((M' \times M') \rightarrow M')) \mid \\ & & \forall x, y, z : M.\ (x\ o\ y)\ o\ z \ = \ x\ o\ (y\ o\ z) \ \in \ M \\ & & \forall x : M.\ (x\ o\ e \ = \ x \ \in \ M) \ \wedge \ (x \ = \ e\ o\ x \ \in \ M)\} \end{array}$$

Here we have actually used a generalization of the set type where we can talk of subsets of tuple types. Such a generalization is easy to define. Using this type, the proposition that the integers under addition form a monoid is simply:

$$\vdash < \texttt{Int}, 0, + > \ \in \ \mathcal{M}$$

7 Hardware Related Projects

7.1 Floating-Point Circuit Verification

We take here the approach of the HOL group [16] of modelling circuits as relations. This approach is sketched at the start of section 6.7. Basin and DelVecchio verified a combinational circuit which was part of a floating-point adder [7]. The circuit consisted of shifting and multiplexing structures.

The author is currently working on verifying that floating-point circuit designs meet their specifications [21]. This work involves much reasoning with parameterized bit-vectors as well as with integer and rational numbers. The IEEE floating-point standard [20] is being formalized within Nuprl's type theory in an abstract way, similar to that of Barrett [4]. The aim of this work is to produce a reusable toolkit of definitions, theorems and tactics that can be used on practical floating-point designs.

7.2 Circuit Synthesis by Extraction

As discussed in section 2.5, theorems in Nuprl's logic can be viewed as specifications of programs. The programs themselves are implicit in the proofs of the theorems and can be extracted out of the proofs. Basin has been using Nuprl to see whether one could adapt this idea so that one could extract circuit designs from proofs of theorems which specify the circuits [5]. He has succeeded in extracting such combinational circuits as barrel shifters and look-ahead carry adders, and is looking into extracting more complex combinational circuits and timed circuits.

7.3 Verification of Hardware Synthesis Systems

This work started with Basin, Brown and Leeser synthesizing combinational circuits from their specifications by using specially tailored rewrite rules [6]. More recently Aagaard verified an implementation of the *weak division* method for simplifying combinational circuits [1]. Weak division is widely used in commercial logic synthesis systems. Aagaard's work is part of a larger project to build a complete formally-verified hardware synthesis system [22].

If formally verified synthesis systems become widely used, then far more designers will benefit from formal methods than just those who use formal methods themselves. There is also no reason why the application of formal methods should be restricted to synthesis systems. One should also be able to verify simulators, timing verifiers, routers and design rule checkers.

8 Discussion / Conclusion

As can be seen in the examples in section 6, type checking can be a powerful organizational paradigm for formal methods. We have shown how type checking neatly partitions off tedious reasoning tasks, such as array bounds checking, so that such tasks can easily be taken of by automated decision procedures. It is also elegant how complete correctness properties of circuits can be phrased concisely as type checking problems.

We have also shown how the expressive type theory enables one to construct precise and natural models of objects one wishes to reason about.

The use of Nuprl's type theory does entail that well-formedness reasoning has to be done using tactics rather than decision procedures. However, well-formedness checking is still automated and well-formedness subgoals are rarely seen explicitly in proofs. With advances in type theory and our work in reflection, we hope to improve the efficiency of well-formedness checking.

We believe that the Nuprl system has several key features which make make it easy to learn and easy to use; a sophisticated X-Windows interface, clear and concise notation for terms, proofs which are readable, and a structured set of proof development strategies.

We realize the need for better user support in terms of tutorial and reference documentation. We also hope in the next year to be able to hold a two or three day workshop for people interested in using Nuprl for formal methods.

9 Practical Details

The Nuprl system consists of approximately 60,000 lines of Common Lisp. Nuprl's tactics and their support functions in ML make up an additional 15,000 lines. Nuprl interacts with X-Windows using the CLX package. We describe in this paper V4.0 of the Nuprl system. This is the current experimental version; we hope to release it sometime in the summer of 1992. The current release version is V3.2.

Nuprl should work on most work-stations running Common Lisp and X-Windows. We currently use it on Sun SparcStation 1 and 2's running Lucid Common Lisp.

Nuprl is in the public domain. Requests for the system or for information can be sent to maxwell@cs.cornell.edu.

10 Acknowledgements

Over the past ten years, over a dozen people have been involved in various ways with Nuprl at Cornell, and the project is indebted to the pioneering work done in LCF and in earlier program verification projects at Cornell.

The hardware related ideas discussed in this paper are the product of joint research between Cornell's Computer Science and Electrical Engineering departments, to apply Nuprl to circuit design.

The author wishes to thank Bob Constable, Jim Caldwell, Miriam Leeser, and Elizabeth Ezra for their comments and encouragement.

References

[1] Mark Aagaard and Miriam Leeser. The implementation and proof of a boolean simplification system. Technical Report EE–CEG–90–2, Cornell School of Electrical Engineering, March 1990. In the*Oxford Workshop on Designing Correct Circuits*, September, 1990.

[2] Stuart 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.

[3] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William B. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual Symposium on Logic and Computer Science*, pages 95–107. IEEE Computer Society, June 1990.

[4] Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989.

[5] David A. Basin. Extracting circuits from constructive proofs. In *International Workshop on Formal Methods in VLSI Design*. ACM, 1991.

[6] David A. Basin, Geoffrey M. Brown, and Miriam E. Leeser. Formally verified synthesis of combinational CMOS circuits. In L. J. M. Claesen, editor, *Formal VLSI Specification and Synthesis*, pages 197–206. North-Holland, 1990.

[7] David A. Basin and Peter Del Vecchio. Verification of combinational logic in Nuprl. In M. E. Leeser and G. M. Brown, editors, *Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, pages 333–357. Springer Verlag, 1990. LNCS 408.

[8] Joseph L. Bates. A logic for correct program development. Technical Report 79-388, Cornell University, Ithaca, NY, August 1979.

[9] Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[10] W. W. Bledsoe. A new method for proving certain Presburger formulas. In *4th International Joint Conference on Artificial Intelligence*, pages 15–21, Tiblsi, 1975.

[11] Robert L. Constable, Scott D. Johnson, and Carl D. Eichenlaub. *Introduction to the PL/CV2 Programming Logic*, volume 135 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1982.

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

[13] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra*, pages 151–184. Springer-Verlag, 1985.

[14] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66, 1985.

[15] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, Cambridge University Computer Laboratory, 1985.

[16] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. North–Holland, 1986.

[17] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[18] F. K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. *IEE Proceedings E: Computers and Digital Techniques*, 133(5):242–254, September 1986.

[19] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988.

[20] IEEE standard for binary floating-point arithmetic. New York ANSI/IEEE Std. 754–1985, August 1985.

[21] Paul B. Jackson. Developing a toolkit for floating-point hardware in the nuprl proof development system. In *Proceedings of the Advanced Research Workshop on Correct Hardware Design Methodologies*. Elsevier, 1991.

[22] Miriam Leeser et al. BEDROC: The Cornell hardware synthesis project. Technical Report EE-CEG-90-6, Cornell School of Electrical Engineering, June 1990.

[23] Per 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.

[24] L. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[25] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[26] Raymond Smullyan. *First Order Logic*. Springer-Verlag, 1968.

[27] M. E. Szabo, editor. *The Collected Works of Gerhard Gentzen*. North Holland, 1969.