January 2001

A User-Level Introduction to the Nuprl Proof Development System

Eric Aaron
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-01-32.

A User-Level Introduction to the Nuprl Proof Development System

Abstract

This document is intended to introduce the key elements of the *Nuprl Proof Development System* (Nuprl, for short) from the perspective of a Nuprl *user*, as opposed to the perspective of someone intimately involved in *developing* or *extending* Nuprl. As such, it may be more appropriate than other Kuprl-related documents for readers who are primarily concerned with uses of Nuprl and not fine details of Nuprl's mathematical foundation. It introduces and illustrates key Kuprl concepts -such as *types, terms, displayforms*, and *tactics* - in the framework of a model of *calculational predicate logic* inference.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-01-32.

A User-Level Introduction to the Nuprl Proof Development System

Eric Aaron
Department of Computer and Information Science
University Of Pennsylvania

Abstract

This document is intended to introduce the key elements of the *Nuprl Proof Development System* (Nuprl, for short) from the perspective of a Nuprl *user*, as opposed to the perspective of someone intimately involved in *developing* or *extending* Nuprl. As such, it may be more appropriate than other Nuprl-related documents for readers who are primarily concerned with uses of Nuprl and not fine details of Nuprl's mathematical foundation. It introduces and illustrates key Nuprl concepts —such as *types*, *terms*, *display forms*, and *tactics*— in the framework of a model of *calculational predicate logic* inference.

1 Preface and Context

The bulk of this document is a (lightly edited) chapter from the doctoral dissertation [2]. Unlike other documents introducing key elements of the *Nuprl Proof Development System* (Nuprl, for short) [8, 12], it was not written by an author whose primary concern was further developing the underlying Nuprl system *per se*. Instead, it was written in the context of a project that applied Nuprl to formally represent a kind of human reasoning. That is, it was written by a *user* of Nuprl, not primarily a developer, for readers more attuned to uses of Nuprl than to fine details of its foundation.

Before presenting this introduction to Nuprl concepts, we present a concise description of its context, the research from which it emerged. The remainder of this section is an extremely brief overview of the ideas in [2]; interested readers are strongly encouraged to read the less-brief overviews in the introductory material of the dissertation itself.

The research explores a new, interdisciplinary approach to cognitive modeling of high-level inference, combining complementary ideas from applied logic, artificial intelligence, and cognitive science. The dissertation describes an application of this approach to a particular inference task: the stylized method for theorem proving in *calculational predicate logic* (described in the undergraduate-level textbook [11]), a variant of classical first-order predicate logic. Theorems are proved in calculational logic by applying a chain of equality-preserving rewrites; for instance, expression $A = B$ could be proved by rewriting A to B or by

Theorem Change of Dummy. Provided $\neg occurs(\text{“}y\text{”}, \text{“}R, P\text{”})$ and function f has an inverse, $(\star x \mid R : P) = (\star y \mid R[x := f.y] : P[x := f.y])$.

Proof. We start with the right side of $(\star x \mid R : P) = (\star y \mid R[x := f.y] : P[x := f.y])$ and show it is equal to the left side.

$$
\begin{array}{ll}
 & (\star y \mid R[x := f.y] : P[x := f.y]) \\
= & \quad \langle \text{One-point rule (8.14)} \\
 & \quad \text{—Quantification over } x \text{ has to be introduced. The One-} \\
 & \quad \text{point rule is the } \textit{only} \text{ theorem that can be applied at first.} \rangle \\
 & (\star y \mid R[x := f.y] : (\star x \mid x = f.y : P)) \\
= & \quad \langle \text{Nesting (8.20) —Moving dummy } x \text{ to the outside} \\
 & \quad \text{gets us closer to the final form.} \rangle \\
 & (\star x, y \mid R[x := f.y] \wedge x = f.y : P) \\
= & \quad \langle \text{Substitution (3.84a)} — R[x := f.y] \text{ must be removed} \\
 & \quad \text{at some point. This substitution makes it possible.} \rangle \\
 & (\star x, y \mid R[x := x] \wedge x = f.y : P) \\
= & \quad \langle R[x := x] \equiv R\text{; Nesting, } \neg occurs(\text{“}y\text{”}, \text{“}R\text{”}) \\
 & \quad \text{—Now we can get a quantification in } x \text{ alone.} \rangle \\
 & (\star x \mid R : (\star y \mid x = f.y : P)) \\
= & \quad \langle x = f.y \equiv y = f^{-1}.x \text{ —This step prepares for the} \\
 & \quad \text{elimination of } y \text{ using the One-point rule.} \rangle \\
 & (\star x \mid R : (\star y \mid y = f^{-1}.x : P)) \\
= & \quad \langle \text{One-point rule (8.14)} \rangle \\
 & (\star x \mid R : P[y := f^{-1}.x]) \\
= & \quad \langle \text{Definition of textual substitution} — \neg occurs(\text{“}y\text{”}, \text{“}P\text{”}) \rangle \\
 & (\star x \mid R : P)
\end{array}
$$

Figure 1: Proof of Theorem Change of Dummy.

rewriting the entire expression to a previously proved theorem.

Calculational logic supports schematic reasoning via metalinguistic operations (such as textual substitution) as well as traditional logical reasoning, all without resorting to a higher-order logic. In particular, it supports reasoning about a general quantifier form —i.e., a general form that can be instantiated into universal quantification, existential quantification, and other quantifier-like accumulation operations (such as sums or products over sets)— that is seen in Figure 1, a calculational proof of a theorem about a familiar property of bound variables. The unusual $(\star \ldots)$ syntax represents that general quantifier form; the variable next to the $\star$ (e.g., x in $(\star x \ldots)$) is bound in that expression. The theorem holds for any instantiation, including both existential and universal quantifiers. (A full explanation of the proof is beyond the scope of this paper. See [2] or [11] for details.)

Because of this metalinguistic character, we interpret calculational logic as a metalogic; theorems such as Change Of Dummy (Figure 1) are meta-level theorems *about* theoremhood in some object-level logic. We occasionally refer to this fact in the sections that follow, but it

is of only secondary importance to our introduction to Nuprl. Readers interested in further details should see [2] or [3].

To implement a Nuprl model of how people perform calculational logic inference, we implemented three distinct levels of mathematics. One level consists of the programs that simulate the inference processes people use — we return to that level shortly. We also implemented two distinct levels of logical language. Calculational logic employs metalinguistic operations; the semantics of metalanguage variables and expressions is described with respect to a lower-level object language. Thus, we implemented the metalanguage that actually appears in proofs like Figure 1 as well as a simple object language to use in implementing the semantics of our metalanguage.

We call our formalized metalanguage the *data language*, because it is the language that people actually use for the calculational logic of [11] — it is the data that guided our implementation. Expressions of the data language may be called *data expressions*. It was not trivial to identify and formalize a data language adequate for the material in [11] that we covered, but details of that process are beyond the scope of this paper. Again, interested readers can see the dissertation [2] or the stand-alone paper [3] for further information.

As part of formalizing a semantics for our data language, we implemented a recursive type *OE* of object language expressions (*object expressions*, for short). Neither the details of type *OE* nor the object language itself is relevant in this paper. Readers should simply understand the role of *OE*: variables (and expressions) in the data language may be of type *OE*.

To further explain the role of metalanguage in our formalization of calculational logic, consider the symbol $\Rightarrow$. It has two meanings in our Nuprl implementation. As a symbol in the data language, it is an *object-language implication constructor*: an operator that, on two arguments of type *OE*, returns an expression of type *OE*. (There are, of course, several *OE*-constructors in the data language, as mentioned in section 2.3.) The other meaning is as standard logical implication on data language expressions: an operator that, given two truth-valued expressions in the data language, returns a truth value. For readers unfamiliar with metalogic, this bears repeating: For one meaning, the symbol stands for a function that returns object expressions; for the other meaning, it stands for a function that returns truth values. (Context disambiguates which meaning of the symbol is intended at any time.) We refer to such metalinguistic constructions later in the paper. Additional explanation of the underlying ideas can be found in [2] and [3], but readers need not fully grasp these ideas to utilize this paper as an introduction to Nuprl.

To implement the actual inference models, we used Nuprl's *tactic* system (based on [10]). The practice of using tactics in automated reasoning allows fully formalized mathematical inference to be expressed and manipulated at a level of abstraction away from primitive logical rules. Tactics are intended to capture high-level inferences, including those that people might naturally make in constructing a proof, and cognitive modeling of logical inference seems like a natural application for tactic-based automated reasoning systems. We apply Nuprl for precisely that reason. (We discuss tactics further in section 3.2.)

The Nuprl proof development system provided a platform for implementing our model of calculational logic inference. This paper is an overview of Nuprl, intended both as an

introduction for the uninitiated and, for readers familiar with Nuprl, a brief summary of the Nuprl features most important for the model in [2]. In the text that follows, we use the word "implementer" to refer to a user of Nuprl, i.e., one (such as the author) who uses Nuprl to formalize and implement a mathematical language and/or system of inference. For history of Nuprl and a more complete introduction, see [8] and [12].

2 Implementing Mathematics in Nuprl

In this section, we introduce some important features of Nuprl that we used in implementing the data language. Our definitions were essentially built from Nuprl's type system and expressed by Nuprl *terms*, so we briefly discuss types and terms. We also introduce Nuprl's system of *display forms*, which preserves the useful distinction between mathematical concepts and notation; for many Nuprl objects, their display forms —which describe how the objects are to be displayed in various contexts— are defined separately from the objects themselves. We exploit display forms in several important ways.

We do not yet consider concepts particular to definitions of calculational logic inference methods or other meta-level programs in this section. We discuss them in the next section.

2.1 Nuprl types

In its standard semantics, Nuprl is based on an intuitionist type theory that is an extension of that of Martin-Löf (see [5, 6, 14]). This has a few significant consequences for us as users and implementers. For instance, booleans and propositions are not the same, and we cannot generally do reasoning by a case split on whether a proposition P is true or false; we need to prove that the truth of a proposition is *decidable* before we can branch on it in an *if-then* context. We intend to model calculational logic, a classical logic. How can this be accomplished in a constructive logic such as Nuprl? The answer is that calculational logic is based around the syntactic property of object-level theoremhood, not the semantic property of object-level truth. Calculational logic is syntactically oriented, and Nuprl is extremely flexible with respect to syntax. Semantic mismatches between the two systems do not affect us.

Indeed, we do not need to consider most of the details of Nuprl's type theory in this introduction. Essentially, we simply defined functions and other mathematical objects in a lambda calculus within a sophisticated type system; that level of understanding should suffice for most of our readers. Nuprl's type theory, however, is much deeper and more broadly applicable than we represent here. See [7] and [8] for more information.

We now discuss some types and related functions, to provide some necessary background and a feel for how we use Nuprl.

Disjoint union A union operation $+$ is one way to combine types: if $T1$ and $T2$ are types, then we can express the notion that a term t is in one of $T1$ or $T2$ by saying it is in the union of the types, i.e. $t \in T1 + T2$. The type system of Nuprl uses a

disjoint union to combine types, so given an element of $T1 + T2$, it must be possible to determine which component t is in, $T1$ or $T2$. To accomplish this, Nuprl uses the term constructors **inl** and **inr**; for $t1 \in T1$, **inl(** $t1$ **)** is in $T1 + T2$, and for $t2 \in T2$, **inr(** $t2$ **)** is in $T1 + T2$. The respective inverse operations are **outl** and **outr**: **outl(inl(t))** is **t**, and similarly for **outr**. We elide the parentheses from **inl**, **outl**, etc. when it improves readability.

We take this opportunity to introduce two ways in which union types are used in our calculational logic implementation. For one, in our type *OE* for object expressions, we conceptually separate object variables from non-object variable expressions; we reflect this by using a disjoint union type of the general form (*variables*) + (other expressions),[1] so we can syntactically determine whether or not any object expression is an object variable. Another use of union types is for a function that looks up values related to keys in a table; we can use a union type (*success type*) + (*failure type*) as the lookup function return type, for any *success type* and *failure type* we may choose. When the lookup succeeds on a key, it returns **inl(** s **)** for some $s \in$ *success type*; when it fails, it returns **inr(** f **)** for some $f \in$ *failure type*.

Cartesian product The expected pair constructor is present: $\langle a, b \rangle \in A \times B$. In contrast, the primitive Nuprl function for pair decomposition may be unfamiliar to readers: the form is **spread(** $p; u, v.b$ **)**, where p is a pair and b is an expression in variables u and v; **spread(** $\langle p, q \rangle; u, v.t$ **)** $= t[p, q/u, v]$. So, for instance, the standard first and second component projections of a pair can be represented as **spread(** $p; u, v.u$ **)** and **spread(** $p; u, v.v$ **)**, respectively.

Dependent types Dependent types are used to create compound types in which one component type depends on a particular value in another component type. For instance, consider a function f on integers that returns an integer on odd inputs and returns a $\mathbb{Z} \rightarrow \mathbb{Z}$ function on other inputs; we would represent the type of f as $x{:}\mathbb{Z} \rightarrow F(x)$, where $F(x) =$ if x is odd then $\mathbb{Z}$ else $(\mathbb{Z} \rightarrow \mathbb{Z})$.

A similar dependent type notion applies to products: if A is a type and B is a type-valued function on A, then an element of $x{:}A \times B(x)$ would be a pair $\langle y, z \rangle$ where $y \in A$ and $z \in B(y)$.[2]

Recursive types Nuprl's type theory can also represent recursive types. For example, consider the recursive structure of unlabeled binary trees with integer leaves; in Nuprl, it can be defined as **rec(** *node*.$\mathbb{Z}$ + *node* × *node* **)**, where variable *node* is bound in the union type expression. Its elements include **inl 5**, **inr <inl 3, inl 7>**, and **inr <inr <inl 2, inl 4>, inl 6>**.

The relationship between a Nuprl type T and its members is expressed with assertions of the form $A \in T$ or $A = B \in T$. $A \in T$ expresses that A is a member of T; $A = B \in T$

[1] We give a full explanation of type *OE* in [2]. We use it here merely as a motivating example.

[2] Other common notations for $x{:}\mathbb{Z} \rightarrow F(x)$ and $x{:}A \times B(x)$ are $\Pi x{:}\mathbb{Z}.F(x)$ and $\Sigma x{:}A.B(x)$, respectively.

expresses that A and B are members of T and equal in T.[3] In this paper, the form $A = B \in T$ refers only to Nuprl equality.

These are just some of the elements of Nuprl's type theory; clearly, it is a very expressive system. All the concepts above are used in our type *OE* of object expressions, but our type definitions do not generally use that much expressive capacity.

In a way, types in Nuprl are the basis for all our work, not just definitions of types needed for implementing calculational logic. Nuprl uses the propositions-as-types correspondence known as the Curry-Howard isomorphism, so its general theorem proving emerges directly from proof rules for its type theory. In addition, Nuprl's lambda calculus —the familiar formalism extended to Nuprl's type system— is the basis for our mathematical definitions. Despite this, readers need not understand most of the concepts in Nuprl's type theory to understand our work in [2]. We generally work with familiar constructs at a level of abstraction away from the type theory (constructs for recursion, case splits, etc.), and we generally explain our work that way.

2.2 Terms and term structure

The Nuprl data structure *term* is used for a variety of purposes. For instance, all Nuprl propositions and expressions in Nuprl's type theory are represented as terms. The mathematical/logical objects we define for our data language —as distinguished from inference methods and other higher-level procedures— are also represented by terms, so we briefly discuss Nuprl terms before presenting the data language implementation.

We do not give a full definition of Nuprl term structure, omitting many details that are not directly relevant to the research in [2]. Our concise description captures the general feel of Nuprl's use of the general-purpose data structure term, and that should be sufficient.

Nuprl terms have roughly the following structure:

(1) $\quad opid(s_1, \ldots, s_n).$

The parts of a term are:[4]

- *opid* is the *operator identifier*; we call this feature an opid. Often, opids serve as the names of terms in our implementation. For instance, the opid of our object-level theoremhood predicate is *Othm* and the opid of our object-level conjunction constructor is *oand*. Readers should generally be unconcerned with low-level details such as Nuprl opids, but we do refer to them in describing our implementation in [2], so we introduce them here.

- s_i is *bound-term* i of the term. Each bound-term itself has a complex structure: $s_j = x_1^j, \ldots, x_{a_j}^j.t_j$, where each of the x's is a *variable* and t_i is itself a term. This

[3] Type enters into the expressions of Nuprl equality because different types may have different equalities. For example, 5 and 10 are equal in $\mathbb{Z}_5$ but not equal in $\mathbb{Z}$.

[4] It is possible to supply other atomic parameters to operators, but we do not generally do so.

bound-term binds free occurrences of variables $x_1^j, \ldots, x_{a_j}^j$ in t_j ; this is the standard notation for variable binding in Nuprl, also used above in contexts such as the pair-decomposition function, **spread(** $p; u, v.t$ **)**.

We discuss terms frequently here and in [2] without using Nuprl term notation to display them. In general, we opt for the common, intuitive notations permitted by the flexible system of Nuprl display forms, described next.

2.3 Display forms

Although Nuprl terms have a uniform syntax, their appearances on page or screen can differ greatly from that syntax. This is one of the strengths of Nuprl: The display forms for a term are defined at a level of abstraction away from the term and its syntax. For instance, consider pair-decomposition operation **spread** mentioned above. Displaying it in uniform term syntax can get somewhat clunky. Instead, it has an abbreviated typical display form: **spread(pair;u,v.body)** is displayed as **let <u,v> = pair in body**, or simply as **(pair/u,v.body)**, according to the user's preferences. This significantly improves readability.

As another example, consider the common propositional logic operators. In a typical Nuprl session, the logical conjunction operator would be input by typing the word "and," corresponding to its opid and therefore its representation in the uniform term syntax presented above. But it is displayed using the character &, a convenient abbreviation; the argument slots are structured so that & appears to be an infix operator in the expected way. Similar mechanisms are used for disjunction, implication, etc., making formulas much easier to read than they would be if logical operations were expressed in the prefix/English term syntax.

This allows for the systematic, unambiguous overloading of notation: we may associate the same display form with several operators, but Nuprl manipulates the unambiguous underlying terms. We exploit this in our work, using (for instance) the symbol $\Rightarrow$ for two different infix operations: the *OE*-constructor for object-level implication and the logical/propositional implication operator in Nuprl. There is no ambiguity when a user enters the expressions into Nuprl —they have different names— but they look the same on the screen. We use display forms to overload traditional logical symbols such as $\Rightarrow$ and $\vee$. This helps us correspond to both conventional Nuprl notation, in which the symbols stand for propositional operations, and the notation in [11], in which they are *OE*-constructors, without any underlying ambiguity. In [2] as in a Nuprl session, context informally disambiguates their usages.

Nuprl users can alter display forms; for instance, a change from & to $\wedge$ could be easily made. This is relevant when considering our display forms for operations relating to quantification in calculational logic, one area where we intentionally diverge from the notation in [11]. For instance, that textbook's notation for the general form of quantification is $(\star X \mid R : B)$, whereas we use $(\star_b X \mid R : B)$, making explicit the indexing argument b. Users who prefer that the b be elided (or perhaps want the star to be a different kind, or other modifications) could make that change. When entering the term, the b would still

need to be accounted for —the term structure of the quantification wouldn't change— but it could simply be erased from the display form. This slightly different display form is not a conceptual divergence from [11]. Indeed, users of our calculational logic implementation in Nuprl could create quantification display forms to suit their tastes.

This system of display forms also permits case-dependent notation, where the same operator can be displayed in different display forms depending on its operand. This has numerous applications, including the ability to elide default values, which we exploit in managing calculational logic quantification. In [2, Chapter 3], we abstractly defined the two expected predicate logic quantifiers $(\forall X \mid R : B)$ and $(\exists X \mid R : B)$ as instances of the general form $(\star_b X \mid R : B)$ for calculational logic quantification, based on whether argument b indicated universal or existential quantification. We concretely implemented that directly in Nuprl using our display form for $(\star_b X \mid R : B)$. When argument b is filled by the constant indicating universal quantification, we simply *display* $(\star_b X \mid R : B)$ as $(\forall X \mid R : B)$. We do not have a separate object for the specific universal quantifier form; it is just a different notation for the same object, given a particular value for b. (Similarly for existential quantification, of course.) When the slot for b is filled by a variable in the general quantification form —say, as part of a lemma statement that quantifies over the possible kinds of calculational logic quantifiers— we use its standard display form, explicitly displaying argument b.

This practice not only makes our implementation more readable, it also encodes the desired relationship between the related quantifier forms. We do not need to define separate operators and prove relationships between them. We have only one operator, corresponding to the definition of our data language, which looks different in different contexts.

There is also a facility in Nuprl for associating user-defined *input commands* with a display form; for instance, we could type `CLquant` to get the general form for calculational logic quantification, no matter what the opid of that operator is. Combining this with the representation of default values, we can directly input operators with certain default values filled in; for instance, we could type `CLall` to get the calculational logic universal quantifier form, i.e., the general form with a particular value filled in for b. It is somewhat similar to the effect of a macro: It is as if we typed `CLquant` and entered a value for b, all by typing `CLall`.

The value of this powerful system in our work is worth noting. In implementing a language that must have the same appearance as a pre-existing notation, the flexibility to assign display forms to specific cases, access them directly, and alter display forms without affecting the underlying mathematics makes our lives as implementers much easier. We can test designs, recover from notation errors, establish shortcuts, use post-hoc mnemonic names, etc., all without significant cost. It permits us all the mechanical and logical benefits of Nuprl's uniform term syntax without constraining the apparent notation of our terms. It truly separates the underlying meaning of expressions from their appearance, a tremendous virtue.

For this reason, we are somewhat loose with some aspects of notation and meta-notation here and in [2]. We may be careless with list notation in our descriptions, for instance, displaying a singleton list as its element or otherwise dropping the brackets that indicate a

list in Nuprl. Such small differences in appearance between our descriptions and our direct inclusions of Nuprl notation should not confuse readers (the typing of expressions will often disambiguate cases when there is some doubt). These differences, after all, are only matters of display, not about the underlying mathematics.

We do not describe how to create Nuprl display forms. Instead, we have discussed only the most important aspects and how they are used in our implementation. For more details on display forms, see [4, 12, 13].

3 Nuprl ML, Tactics, and Proofs

Inference modeling uses a different set of tools from those used for formalizing the mathematical language of calculational logic. Nuprl inferences (such as those used in implementing calculational logic inference) are at a meta-level to the mathematics formalized using the concepts introduced in section 2. To implement inferences and various meta-level auxiliary functions, Nuprl uses a dialect of the programming language ML —the metalanguage of the Edinburgh LCF system (see [10])— as its general-purpose metalanguage. Once we have introduced *Nuprl ML* and the related concepts of *tactics* and *tacticals*, we will discuss a few details of Nuprl inferences and proofs.

As with section 2, this is a very brief overview of an intricate system. For a more balanced and thorough introduction to the use of metalanguage in Nuprl, see [1, 8, 12].

3.1 Nuprl ML

We use Nuprl ML for two distinct but complementary purposes: creating tactics for carrying out inferences and creating auxiliary functions for doing general tasks that are not well-suited for expression as inferences, such as

- simple list manipulation,
- heuristic guessing (without proof) whether two expressions might be equal,
- heuristic guessing (without proof) the type of an expression in an environment.

The second and third examples above reflect that we may want to use heuristics that are not as computationally expensive as tactics to determine if a tactic is even worth calling in a context. Running a small, non-tactic ML program is often far less time-consuming than running a tactic, so we can use ML to implement heuristics to guide tactic inferences.

Although tactics are ML programs, we tend to consider them separately from general ML programs that do not involve tactics. Indeed, we may use the phrase "ML programs" to refer to only those non-tactic programs. In this section, we discuss the ML language in general, independent of any explicitly tactic-related constructs. We discuss tactics in section 3.2.

Declarations d

d ::=	`let` b	ordinary variables
|	`letrec` b	recursive functions

Bindings b

b ::=	p=e	simple binding
|	$p_1\ p_2 \ldots p_n$ = e	function definition

Patterns p

p ::=	*var*	variable
|	p_1.p_2	R list cons
|	p_1,p_2	R pairing
|	[p_1;$p_2 \ldots$;p_n]	list of n elements (n may be 0)

Expressions e

e ::=	*ce*	constant
|	*var*	variable
|	$e_1 e_2$	L function application
|	e_1.e_2	R list cons
|	e_1@e_2	R list append
|	e_1=e_2	L equality
|	`not` e	negation
|	e_1&e_2	R conjunction
|	e_1,e_2	R pairing
|	`failwith` e	failure with string
|	`if` e_1 `then` e_2 `else` e_3	conditional
|	e_1 ? e_2	failure trap
|	e_1;$e_2 \ldots$;e_n	sequencing
|	[e_1;$e_2 \ldots$; e_n]	list of n elements (n may be 0)
|	d `in` e	local declaration
|	\ $p_1 p_2 \ldots p_n$. e	abstraction

Figure 2: ML syntax equations

In Figure 2, we present a subset of the ML syntax given in [1, section 3.1]. See [1] for more details on both these syntax equations and their associated semantics.

To make this section a somewhat more self-contained introduction, here are a few key points:

- It may be difficult to see how the local declaration construct (given under **Expressions**) is used, so we provide an example. Consider this contrived definition of a factorial-like function f :

  ```
  letrec f n =
     let non_zero i = (i > 0) in
        if non_zero n then n · (f n − 1) else 1
  ```

  (We ignore any issues about restricting values of n .) This illustrates the local declaration of function **non_zero** in the declaration of function f as well as the constructs for function application and recursive function declaration.

- Only functions can be defined with `letrec`. For example, `letrec x = 2-x` is syntactically incorrect.

- All the variables occurring in a pattern must be distinct. On the other hand, a pattern can contain multiple occurrences of the wildcard '`()`'.

3.2 Tactics

As mentioned in section 1, the *tactic* structure of Nuprl (based on [10]) is one of the primary reasons it seemed practical to model cognitive inference in an automated reasoning system. (See [9] for elaboration on the idea of tactics used in Nuprl and their representation of mathematical thinking.) Given a collection of pre-specified primitive inferences, a tactic is a program for reducing a proof goal to premises by iteration of these primitive inferences. Essentially, a tactic is a program for constructing such an inference tree; which tactics are applied depends on how one wants to generates subgoals from the proof goal.[5] Executing a tactic gives rise to an inference step; the premises of that inference are the unproved leaf-premises of the primitive proof tree.

The execution of a tactic might raise an exception or fail to terminate. For a program to count as a tactic, however, if it terminates without exception, it must be guaranteed to generate subgoals justifiable by the primitive rules.

In order to freely write heuristic tactics, there must be some practical criterion for recognizing tactics. In ML, this criterion is type checking to the type of tactics.

The labor of justifying inference is split between justifying "tactichood" and checking the primitive inference rules. (Presumably, the primitive inference rules are formulated to make such verification feasible. For example, they tend to be schematic.) The justification

[5] We discuss tactics from a more cognitive science-oriented perspective in the conclusion of [2].

of complex forms of inference via tactics is in terms of the generic criterion for being a tactic and is not schematically described.

Here are a few simple tactics in Nuprl that pertain to points of interest in our implementation:

- **Id** is the identity tactic; applying it to a proof goal leaves it unchanged. As we illustrate shortly, it is useful in constructing complex tactics from simple ones.
- **FailWith** takes a character string as an argument. When applied to a proof goal, it fails and outputs its string argument as an error message. In conjunction with a failure-trapping mechanism, it permits implementers to output more informative, customized failure messages than Nuprl's generic ones.
- **AUTO** is perhaps the most important tactic in Nuprl. It performs a wide variety of simple inferences, such as typing judgments, some trivial equalities, and some trivial proofs from hypotheses. In essence, **AUTO** represents the class of inferences considered too obvious for Nuprl's users.

  In implementing calculational logic inference, we did not use **AUTO** in our tactics. It is very high-level, and tactics built around it can be fragile because someone may want to change **AUTO** and the class of obvious inferences it represents without affecting other, more specific tactics. Therefore, we presume that users of our calculational logic tactics will invoke **AUTO** themselves.
- There are several varieties of chaining lemmas in Nuprl. For instance, **BackThruLemma** is a simple backchaining tactic, whereas **BackThru** is a more complex one, wrapping **BackThruLemma** in levels of pattern-matching and other mechanical intelligence. There are also other tactics for forward chaining, backward chaining, and lemma instantiation, and we use some varieties of them extensively.

  For reasons similar to those expressed regarding **AUTO**, above, we used only the lowest-level chaining tactics in our calculational logic implementation. Indeed, higher-level chaining tactics may themselves call **AUTO**, so the identical reasons apply.

Although all of Nuprl ML can be used to create new tactics from old ones, for common applications, it is often simpler to use Nuprl's language of *tacticals*, which are functions for composing tactics. We made extensive use of tacticals in creating our calculational logic tactics, but their use as programming constructs is so simple that we do not spend much space here even introducing them. Here are a few simple examples of tacticals; see [12] for a more complete list.

- **REPEAT** is analogous to the common programming language looping construct: for tactic T, **REPEAT** T repeatedly runs T on the subgoals resulting from previous applications until no progress is made.
- **ORELSE** is the failure-trapping tactical: for tactics $T1$ and $T2$, $T1$ **ORELSE** $T2$ tries running $T1$; if it fails, it runs $T2$ instead. With the **FailWith** tactic mentioned above, this can be used for customizing error messages.

- **Try** T tries to run tactic T on a proof goal, but if T fails, it leaves the proof goal unchanged; it is defined as T **ORELSE Id**. This is extremely useful in tactic development.
- $T1$ **THEN** $T2$ first runs $T1$ and then runs $T2$ on all subgoals generated by $T1$. The combination of a tactic T with **AUTO**, written as T **THEN AUTO**, has an abbreviated alternative display form: T ..... This abbreviated form comes up in example (3) in section 3.3 below.
- **Complete** T is useful for determining if tactic T will finish a proof; it runs T and fails if T generates any subgoals.

Tactics can get increasingly complex; there is a full programming language for use in constructing them. Some complex inference patterns could also be captured in lemmas; a lemma can represent the result of a chain of inferences just as a tactic can. In this way, there is an interesting division of inferential labor, as it were, between tactics and lemmas: increased reliance on one of them can facilitate less dependence on the other. In our implementation, we tended to use lemmas where it seemed helpful, rather than create more complicated tactics. This was, however, simply a design decision. If our important inferences had been more clearly expressed by tactics than lemmas, we would have decided differently.

3.3 Proving things in Nuprl

Nuprl's inference style is based around *sequents*. A sequent is written as $H_1, \ldots, H_n \vdash C$, where C is the *conclusion* of the sequent and each H_i is either a *hypothesis* or a *declaration* of a variable with its type. Normally, Nuprl displays sequents vertically, with explicit numbers for the hypotheses:

(2)
$$\begin{array}{l} 1.\ H_1 \\ \vdots \\ n.\ H_n \\ \vdash C \end{array}$$

We may refer to the hypotheses and the conclusion of a sequent as *clauses*. In addition, by *goal* (or *proof goal*); we may refer to either a full sequent or only its conclusion.

Our introduction to Nuprl proofs is a simplification of the more detailed one in [12] and other sources, intended primarily to permit readers to understand [2]. We see proofs as tree structures in which every node has a sequent component and a tactic component; the tactic component of a node may be empty or otherwise ill-formed. The children of a node N are the subgoals generated by the tactic of N applied to the goal of N . If a node has no children, its tactic fully solves its proof goal.

As an example of how tactics generate subgoals, consider the following contrived example for backchaining:

(3)
```
1. propn1 : Prop
2. propn2 : Prop
⊢  ¬(propn1 ∨ propn2)
by  BackThru: Thm* ∀A,B:Prop. ¬A & ¬B ⇒ ¬(A ∨ B) ....
 \
  ⊢  ¬propn1  by  <TACTIC>
  ---
  ⊢  ¬propn2  by  <TACTIC>
```

It backchains through a simple lemma, then calls tactic **AUTO** —which is represented in abbreviated form by the four dots after the backchained lemma— to handle routine manipulation (proving typing subgoals, etc.). Note that antecedents in the lemma became subgoals after the tactic.

In the expected way, a proof is complete when all of its nodes have the expected properties: all variables in every sequent are bound in that sequent; all nodes have a tactic; every goal is proved by its tactic, assuming provability of its children; etc.

Proof goals in Nuprl also have a *label*, which roughly indicates their classification or purpose. For instance, goals corresponding to the primary inferences generally have the (normally elided) default label **main**; other labels, such as **assertion** and **rewrite subgoal**, mark proof goals that arise in particular circumstances with which most readers of [2] need not be concerned. As implementers, we must take some minimal care to manage our tactics correctly on these labels. With one exception, discussed immediately below, users of the system never encounter them.

The one kind of non-**main** proof goal that is significant to users of our calculational logic system is *well-formedness* goals, which are used to establish that Nuprl expressions have the necessary types; they have the label **wf**. Well-formedness goals are critical and pervasive in Nuprl, because everything must be type-correct. They are typically handled automatically by Nuprl's **AUTO** tactic. Often, when Nuprl users want to invoke a tactic *TAC*, they wrap it in **AUTO**, instead applying ***TAC*** **THEN AUTO**. It is a strength of the tactic system of Nuprl —and its **AUTO** tactic in particular— that Nuprl users are often insulated from typing judgments and other simple inferences. Further, when **AUTO** does not solve a well-formedness subgoal, it is often a useful indication of user error; when a user fails to assign a correct type to a variable, for instance, it comes up in the form of an unsolved **wf** goal.

In our implementation of calculational logic inference, we assume that our tactics will be wrapped in tactic **AUTO**, so we do not try to solve well-formedness subgoals ourselves. We do solve all other subgoals that may emerge, no matter what their label, but users of our calculational logic tactics who do not also call **AUTO** may well be greeted with a message from Nuprl informing them that there are hundreds of unsolved subgoals remaining. In our experience, these are all **wf** subgoals. Well-formedness subgoals that remain after **AUTO** have the same value as in any other Nuprl context: they frequently indicate user error. Our tactics do nothing to obscure this.

4 Concluding Remarks

As mentioned in this document, specific features of Nuprl guided our design decisions in several ways. Prominently, Nuprl's type theory influenced our choice for type *OE*, which affected the rest of our implementation. Nuprl's many degrees of abstraction —between syntax and display, between term-level and meta-level, etc.— also affected the overall structure of our mathematics and our implementation of calculational inference, as did the relative simplicity of expressing some procedures as tactics and others as general ML programs.

We conclude this paper with a few comments on how we used tactics. One of the primary aspects of our design philosophy for tactics has already come up in our discussion of `AUTO`, but it bears repeating: It was our conscious goal to keep our tactics as low-level as possible, to build them from component tactics that are themselves as close as possible to Nuprl's primitive rules. One result of this is that our tactics never called `AUTO` directly, but we also never used high-level tactics for chaining, expression decomposition, etc. This results in a more robust system, because Nuprl's high-level tactics are more likely to be changed from version to version (or customized from user to user) than low-level ones, and our dependence on these more variable tactics is minimized.

In addition, because our implementation is part of a feasibility demonstration for our overall cognitive modeling method, we frequently used an exceptional tactic that is not mentioned on the list in section 3.2: `Fiat` is a tactic that, when applied, signals Nuprl to treat an incomplete proof as if it were complete. Indeed, it embodies the idea of "proving by fiat," resulting in Nuprl's accepting a proof goal as fully solved without further justification. `Fiat` could even prove `False`, which makes it extremely dangerous to use. We used it regularly to speed up our implementation of calculational logic inference — instead of spending time proving obvious properties of our implementation, we simply stated them as lemmas and used them in our tactics, a philosophy we refer to as *state-and-use* in [2]. If there are errors in our implementation, they probably arise from gaps in reasoning that are artificially filled with `Fiat`.

In fact, we nearly overlooked such an error. We created a heuristic ML program `termeql` to tell if two Nuprl terms are equal modulo a simple equivalence relation that we did not want to take the time to implement formally in Nuprl; then, we wrote a tactic that essentially claimed that if we wanted to prove a goal $A = B \in T$, the truth of `termeql` A B was sufficient. Instead of formally working through all the reasoning for such a tactic, we used `Fiat`, and that resulted in a false tactic: It failed to account for all the typing information needed to prove such a goal. We believe that no such errors currently exist in our system, but as a matter of full disclosure, we felt compelled to mention their possibility due to our use of `Fiat`.

Acknowledgments

The author thanks Stuart Allen and Bob Constable for their advice in shaping this paper.

References

[1] Nuprl ML manual, 1993. No author attributed.

[2] E. Aaron. *Tactic-based modeling of cognitive inference on logically structured notation.* PhD thesis, Cornell University, 2000.

[3] E. Aaron and S. F. Allen. Justifying calculational logic by a conventional metalinguistic semantics. Technical Report TR99-1771, Department of Computer Science, Cornell University, 1999.

[4] S. F. Allen. From dy/dx to []P: a matter of notation. In *Proceedings of the 4th Workshop on User Interfaces for Theorem Provers, UITP-98*, Eindhoven, Netherlands, 1998. Lawrence Erlbaum Associates.

[5] S. F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 215–224. IEEE, June 1987.

[6] S. F. Allen, R. L. Constable, D. J. Howe, and William E. Aitken. The semantics of reflected proof. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 95–105, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[7] R. L. Constable. Types in logic, mathematics and programming. In S. R. Buss, editor, *Handbook of Proof Theory*, pages 683–786. Elsevier, 1998.

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

[9] R. L. Constable, T. Knoblock, and J.L. Bates. Writing programs that construct proofs. *J. Automated Reasoning*, 1(3):285–326, 1984.

[10] M. Gordon, A. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1979.

[11] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag, New York, 1993.

[12] P. Jackson. The Nuprl proof development system reference manual and user's guide, 1994. http://cs.cornell.edu/Info/Projects/ NuPrl/manual.with.index/it.html (May, 2000).

[13] C. Mannion and S. F. Allen. A notation for computer aided mathematics. Technical report, Cornell University, Ithaca, NY, 1994.

[14] P. Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress of Logic, Methodology, and Philosophy of Science*, New York, 1982. North-Holland Publishing Co.