Nuprl as a General Logic*

Robert L. Constable
Douglas J. Howe

TR 89-1021
June 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This research was supported in part by NSF grant CCR-8616552 and ONR grant N00014-88-K-0409.

Nuprl as a General Logic*

Robert L. Constable *Douglas J. Howe*†

Abstract

Study of the architecture and design of proof development systems has become important lately as their use has spread and become closely tied to programming environments. One of the central issues is how to provide a general framework for defining and using a variety of logics in such systems; in particular, whether it is better to start with a simple core system, such as the typed λ-calculus with dependent function types, or start with a very rich theory providing a formalized metatheory. The first approach is exemplified by the Edinburgh LF. Here we illustrate the second approach by showing how to use Nuprl as a framework for defining logics in the style of the LF. Central to the viability of the second approach is a method of showing that the encoding of user defined logics in Nuprl is faithful. This paper presents a new semantic method to solve the problem. The method is applicable to the LF as well, and seems simpler than the syntactic methods that previously were used and which seem to hinder the use of rich theories as logical frameworks.

1 Introduction

It is known that the lambda calculus with dependent function types models the deductive apparatus of numerous logical systems [3,7]. There are proposals for using such a calculus as the basis of a computer system for

*This research was supported in part by NSF grant CCR-8616552 and ONR grant N00014-88-K-0409

†Authors' address: Department of Computer Science, Cornell University, Ithaca NY 14853

1.1 Overview of the Syntactic Method

The LF approach to defining logics is to present the syntax of the language and the inference rules in terms of typed constants. Type constants represent the syntactic categories of the language, and term or formula constructors are represented by constants that are typed as functions over these type constants. For example, the implication connective builds a new proposition, $\supset(p)(q)$, from propositions p and q. If the type of propositions is denoted o, then $\supset$ has the type $o \rightarrow o \rightarrow o$. Next, judgements of the form $H \vdash G$, meaning that G is provable from H, are interpreted as types, $H \rightarrow G$. The universal judgement that G is provable for all objects x in the type i of individuals is interpreted as a *dependent* function type $x:i \rightarrow G$. Another notation for this type is $\Pi\, x:i\,.\, G$. Rules of proof are also typed constants. For example, one can consider the type $\mathit{true}(p)$ for any $p \in o$ to be the type of all proofs of the proposition p. Then the implication introduction rule can be defined by the constant $\supset I$ of type:

$$p:o \rightarrow q:o \rightarrow (\mathit{true}(p) \rightarrow \mathit{true}(q)) \rightarrow \mathit{true}(\supset(p)(q)).$$

According to this approach, a proof of a proposition such as $p \supset (q \supset p)$ will be given by an expression built up from rule constants and having type $\mathit{true}(p \supset (q \supset p))$.

1.2 Implementations

For many kinds of inference system it is a straightforward exercise to write down the syntax and proof rules in a language providing dependent function types. Systems having such a language include Automath, Nuprl, and Griffin's implementation of LF. The interesting point is whether this definition can be used by the system to present a useable implementation of a logic. Automath does not provide an interactive environment, so the question does not apply. The implementation of LF essentially provides only an editor. We believe that it is an interesting point that the Nuprl editing and inference mechanisms apply directly to the logics defined in the manner of LF. Thus Nuprl is not only an environment for a specific built-in logic, namely constructive type theory, but also an environment for logics defined in it.

2 The Classical Predicate Calculus in Nuprl

2.1 Background

Nuprl is a large system, but only a small fragment is needed to understand its role as a logical framework. First, the dependent function space construction is written as $x : A \rightarrow B$ (although later we will switch to the more familiar $\Pi\, x{:}A\,.\,B$). When x does not occur in B, the ordinary function space construction $A \rightarrow B$ can be used. The definitions and theorems that define the predicate calculus are displayed in a *library*. A library can contain three kinds of objects: definitions, theorems and tactics.

Definitions are used to build a surface syntax. For example, we can make a definition that will allow applications of the implication constant to appear as $P \supset Q$. Theorems have the form `>>` *type-expression*. Thus in Nuprl a proposition is a type. Proofs of theorems give rise to objects, called *extractions* or *extracted terms*, that inhabit the type. These objects can be referred to later in the library via the term `term_of(`*name*`)`, where *name* is the name of the theorem.

The inference rules of Nuprl deal with *sequents*, which are objects of the form

$$H_1,\ H_2,\ \ldots,\ H_n \gg P$$

where P is a type and where each H_i is either a type or a variable declaration of the form $x : T$ for x a variable and T a type. The H_i are referred to as *hypotheses*, and P is called the *conclusion* of the sequent.

To define the predicate calculus in Nuprl, we construct a *context*, *i.e.* a collection of declarations that certain constants have certain types. We do this by constructing a sequence of theorems whose statements are types and whose extractions are the constants we want declared.[1] We then direct Nuprl to assume that these theorems are proven. For example, the theorem corresponding to the declaration of the implication constructor has statement `>> o->o->o` and extraction a constant that is displayed as $\supset$. A future version of Nuprl will provide for a more direct and structured treatment of contexts.

[1]The declaration could be made more direct, but it is convenient to treat the constants as extractions.

otherwise, a subgoal is generated that has conclusion `true(`B`)` and as a new hypothesis `true(`A`)`, where A and B are obtained by matching. Other subgoals are generated, but they are all *membership* subgoals (e.g., to show that $A \in$ `o`) and such subgoals are proved automatically by the tactic `Autotactic` which is invoked after most top-level tactic invocations (an application of the autotactic is via a definition and is indicated in a proof tree by "..."). Note that in general type-checking is not decidable in Nuprl. This is not so significant here since the autotactic can, in practice, deal with the bulk of the typechecking requirements, and in particular can prove any typing subgoal that falls within the Nuprl subset corresponding to LF.

For the constant ⊃E for implication elimination we have the theorem

```
>> P:o -> Q:o -> P⊃ Q -> P -> Q,
```

and corresponding tactic

```
let imp_E = make_elim_rule 'imp_E_' [] ;;.
```

The tactic `imp_E` takes an integer argument i, and attempts to match the pattern `true(`$A \supset B$`)` against the i^{th} hypothesis of the sequent. If successful, two non-membership subgoals are generated. One has the same conclusion and `true(`B`)` as a new hypothesis, and the other has `true(`A`)` as the conclusion.

For the existential quantifier constant ∃, we have `>> (i->o)->o`, and for its elimination rule:

```
>> P:(i->o) -> Q:o -> ∃P -> (x:i -> P(x) -> Q) -> Q
let some_E = make_elim_rule 'some_E_' [] ;;.
```

Here are some theorems proved in this context:

```
>> P:o -> Q:o -> P⊃ (Q⊃ P)
>> P:(i->o) -> Q:(i->o) -> a:i ->
            (∀x. P(x)⊃ Q(x)) ⊃ P(a) ⊃ Q(a)
>> P:(i->o) ->  ¬∀x. ¬P(x) ⊃ ∃x. P(x)
```

We show the second of the three steps of the proof of the first of these theorems. Note that the construction of the predicate calculus proof, in terms of the constants declared in the context, is completely implicit.

Open-endedness

The usual semantics of Nuprl's type theory is constructed roughly as follows. For a more detailed account see [2] and [1]. We start with a collection $\mathcal{T}$ of closed terms. The notion of *type* does not enter at this stage; there is only one syntactic category, so that any term constructor can take arbitrary terms as arguments. The term constructors include those of the untyped λ-calculus, various data constructors (such as pairing), and constructors such as $\Pi\, x : A\,.\, B$ that will be used to form type expressions. Next we define an *evaluation* relation on $\mathcal{T}$. We will write $a \leftarrow b$ when a is the value of b (that is, b evaluates to a). Evaluation has the property that if $a \leftarrow b$ then $a \leftarrow a$. Restricted to the untyped λ-calculus, evaluation corresponds to weak head-reduction.

The final stage in the construction of Nuprl's semantics is to build a *type system* by picking a collection of terms to denote types and associating with each such term a set of terms and an equivalence relation on the set. These types are defined with an inductive construction, and terms are put into types if they have the appropriate behaviour under evaluation. For example, if $Int \leftarrow T$ then T is defined to be a type whose members are all terms t such that $n \leftarrow t$ where n is one of Nuprl's constant terms representing an integer. Also, if the terms A and B have been defined as types, then the term $A \rightarrow B$ will also be a type, and a term t will be a member of $A \rightarrow B$ exactly when (ignoring equality) for every term a in the type A the term $t(a)$ is in the type B.

One of the important points here concerning Nuprl's inference system is that it reflects the idea that all that is important about functions is their extensional character; in particular, if $f \in A \rightarrow B$ then it can be inferred that for every $a \in A$, $f(a) \in B$, but *not* that f must evaluate to a term of the form $\lambda x.\, b$. This allows us to extend Nuprl's set of terms and evaluation relation to incorporate constants representing new functions. Another important point is that although Nuprl has a type U_1 which denotes the collection of all ("small") types, none of Nuprl's rules restrict U_1 to contain only the types put in it by the semantics. This allows us to extend the semantics with an arbitrary collection of new base types.

We now proceed to construct an extended Nuprl semantics based on a given collection of new base types and functions over these types. Nuprl has a large collection of type constructors, most of which are uninteresting

- *true* $\leftarrow b$, and either $0 \leftarrow c$ and a is *False* or $1 \leftarrow c$ and a is *True*.

A *type system* over $\mathcal{T}$ is a partial function σ from $\mathcal{T}$ to the set of partial equivalence relations[2] over $\mathcal{T}$ such that $\sigma(T)$ is defined if and only if there is a $T' \leftarrow T$ where $\sigma(T')$ is defined, and such that if $\sigma(T)$ is defined then $\sigma(T)(a, b)$ if and only if there are $a' \leftarrow a$ and $b' \leftarrow b$ with $\sigma(T)(a', b')$.

Suppose σ is a type system over $\mathcal{T}$. Inductively define a type system $\sigma' = \mathcal{F}(\sigma)$ as follows.

- If *True* $\leftarrow T$ then $\sigma'(T) = \{ \langle a, b \rangle \mid r \leftarrow a, b \}$
- If *False* $\leftarrow T$ then $\sigma'(T) = \emptyset$.
- If $\kappa_A \leftarrow T$ for some $A \in \mathcal{A}$, then $\sigma'(T)$ is the set of all $\langle a, b \rangle$ such that $\kappa_x \leftarrow a, b$ for some $x \in A$.
- If $\Pi\, x\!:\!A\,.\,B \leftarrow T$ where $\sigma'(A)$ is defined, and where for every a and a' such that $\sigma'(A)(a, a')$, $\sigma'(B[a/x])$ and $\sigma'(B[a'/x])$ are defined and equal, then $\sigma'(T)$ is

$$\{ \langle b, b' \rangle \mid \sigma'(A)(a, a') \Rightarrow \sigma'(B[a/x])(b(a), b'(a')) \}.$$

- Otherwise, if $\sigma(T)$ is defined then $\sigma'(T) = \sigma(T)$.

We can now define our desired type system by iterating $\mathcal{F}$. Define $\sigma_1 = \mathcal{F}(\emptyset)$. Suppose σ_n is defined. Define $\sigma(T)$ to be the set of all $\langle A, A' \rangle$ such that $\sigma_n(A)$ and $\sigma_n(A')$ are defined and equal, if $U_n \leftarrow T$; $\sigma(T)$ is $\sigma_n(T)$ otherwise. Define $\sigma_{n+1} = \mathcal{F}(\sigma)$. The type system we want is $\tau = \bigcup_n \sigma_n$. It is straightforward to show that τ is in fact a type system and that $\tau = \mathcal{F}(\tau)$. We say that a term T is a type if $\tau(T)$ is defined, and that two types T and T' are equal if $\tau(T) = \tau(T')$. When T is a type we write $a \in T$ and $a = a' \in T$ for $\tau(T)(a, a)$ and $\tau(T)(a, a')$ respectively. We can prove that if T is a type then $T \in U_i$ for some i, and if $A \in U_i$ and $B \in A \rightarrow U_i$ then $(\Pi\, x\!:\!A\,.\,B) \in U_i$.

For $S \in \tilde{\mathcal{A}}$, define the term $rep(S)$ as follows. For $A \in \mathcal{A}$, $rep(A)$ is κ_A, and $rep(S_1 \rightarrow S_2)$ is $rep(S_1) \rightarrow rep(S_2)$. (Note that we use the same notation here for the set-theoretic function space and the Nuprl function type; we similarly overload "$\in$"). The important properties of the type system τ are given in the following three theorems.

[2] A partial equivalence relation is a transitive symmetric relation.

of $\mathcal{C}[true(\tilde{p})]$ in the type system τ. We show that p is true in the model M of Th over A by "instantiating" the context $\mathcal{C}[\cdot]$ with particular terms of $\mathcal{T}$. With the variable i we associate the term κ_A; with o, κ_{Bool}; and with the variable *true* we associate the constant *true*. We will only give a few examples of how to associate terms with the other components of $\mathcal{C}[\cdot]$. With $\exists$ we associate $\kappa_{f_\exists}$ where $f_\exists \in (A \to Bool) \to Bool$ is a function such that $f_\exists(g) = 1$ if and only if $g(a) = 1$ for some $a \in A$. By Theorems 2 and 3

$$\kappa_{f_\exists} \in (\kappa_A \to \kappa_{Bool}) \to \kappa_{Bool}.$$

Finally, with $\exists E$ we associate $t = (\lambda\, p\ q\ x\ g.\ r)$. By Theorem 2 we have

$$\begin{aligned} t \ \in\ & \Pi\, p{:}\kappa_A \to \kappa_{Bool}.\ \Pi\, q{:}\kappa_{Bool}.\ \Pi\, x{:}true(\kappa_{f_\exists}(p)).\\ & \Pi\, g{:}\,(\Pi\, x{:}\kappa_A.\ true(p(x)) \to true(q))\ .\ true(q). \end{aligned}$$

This can be seen as follows. Suppose there are closed terms p, q, x, and g such that $p \in \kappa_A \to \kappa_{Bool}$, $q \in \kappa_{Bool}$, $x \in true(\kappa_{f_\exists}(p))$ and

$$g \ \in\ \Pi\, x{:}\kappa_A.\ true(p(x)) \to true(q).$$

Since the type $true(\kappa_{f_\exists}(p))$ has a member, we must have that $\kappa_1 \leftarrow \kappa_{f_\exists}(p)$, and so there must be some $a \in \kappa_A$ such that $\kappa_1 \leftarrow p(a)$. The term $g(a)(r)$ is thus a member of $true(q)$, and so r must also be a member.

For each of the variables in the context $\mathcal{C}[\cdot]$ we have associated a term of the correct type, and so by applying the term t successively to all of these terms we get a member of

$$true(\tilde{p}[\kappa_A/i, \kappa_{Bool}/o, \ldots]).$$

This means that $\tilde{p}[\kappa_A/i, \ldots]$ must evaluate to κ_1. Since the terms we associated with the logical connectives in $\mathcal{C}[\cdot]$ represent the usual truth functions for first-order logic, it follows that p is true in M. For example, if p is $\exists\, x.\ q(x)$, for q a predicate symbol of Th[3], then

$$\kappa_1 \leftarrow (\kappa_{f_\exists}(\lambda x.\, \kappa_{f_q}(x)))$$

[3]We have not included function or predicate symbols in Th, but doing so presents no difficulty.

Extraction. Suppose we are defining a formal system $\mathcal{F}$ whose basic form of judgement is $t \in T$ for terms t and types T. If one is using the type system of $\mathcal{F}$ to represent logic, then one will often wish to show that a type has a member, and in doing so it would be convenient not to have to provide the explicit construction of the term. This can be accounted for in Nuprl by deriving new forms of the rules to deal with assertions of the form

$$\Sigma\, t\!:\!term.\ t \in T.$$

To construct a proof of T where the term construction is implicit, one states the goal in the above form and uses the new forms of the rules (definitions can also be used hide the references to t). From the resulting Nuprl proof can be extracted a pair whose components are a term t of $\mathcal{F}$ and the $\mathcal{F}$-proof that $t \in T$.

References

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

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

[3] A. Avron, F. A. Honsell, and I. A. Mason. Using typed lambda calculus to implement formal systems on a machine. Laboratory for the foundations of computer science technical report, Edinburgh University, 1987.

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

[5] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[6] T. G. Griffin. An environment for implementing formal systems. Technical report, Computer Science Department, Cornell University, 1987.