An Abstract Semantics for Atoms in Nuprl*

Stuart F. Allen
Dept. of Computer Science
Cornell University

2006

Introduction

With the standard inference rule set for Nuprl, the type Atom cannot be proved either to be finite or infinite, despite the fact that any character string (over a certain finite alphabet) can be used to form a canonical expression for a member of the type. For each $k \in \mathbf{N}$ one can prove in the logic a formula to the effect that there are at least k atoms. One can do this simply by exhibiting k distinct character strings as constants. But one cannot prove a statement to the effect that for all $k \in \mathbf{N}$ there are at least k atoms. Similarly, one cannot in the logic provide an enumeration of the Atoms, nor show that there are finitely many Atoms.

This gap in provability may seem strange in a logic, though it is not so strange as a way of treating some data values as purely atomic. For example, the tokens that make up members of enumeration types in Wirth's well-known language PASCAL cannot be analyzed either, as opposed to the atoms of some LISP dialects which have explode and implode operations that convert between atom values and character strings.

A pragmatic reason for omitting such rules and operations from a logic might be to provide a convenient abuse-proof facility for treating a class of values as atomic. Our purpose here is to explain a principled semantic treatment of atoms that characterizes these expressive limits and actually invalidates the rules that would allow for demonstrating in the logic either finiteness or infiniteness of the atoms.

Our approach here will be strictly semantic rather than ontic. An ontic approach might attempt to define a class of objects that are somehow impervious to reasoning by means of the rules we wish to invalidate. Perhaps one would stipulate some kind of ur-element or define some open-ended always finitely-extensible class of values. That is not the aim here.

We will provide a "supervaluation" semantics that stipulates truth of assertions in the logic in terms of quantification over plenty of choices for interpreting the class of atoms and the strings denoting them. Note that this is not a matter of distinguishing

*This work was supported in part by the National Science Foundation under Grant No. 0208536.

between constructive and classical logic pertaining to the excluded middle; indeed, supervaluation semantics have been used in explications of how excluded middle can be maintained even when vague predicates are involved, which appear to violate excluded middle.

We will characterize a semantics based upon computational methods whose use of atom constants is limited to testing identity between them and "passing them around."

This semantics will then be further exploited to justify a new inference rule, valid under the supervaluation semantics, which allows the atom constants appearing throughout a premise to be replaced by other atom constants through a 1-1 function.

Nuprl Semantics and Logic for Computational Type Theory

While the semantic method may be applied to other logics, we develop it here for Nuprl. Nuprl logics are sequent logics for building proofs of assertions whose semantics are given in various ways, designed for providing automatic assistance to users in successively reducing claims down to sufficient subgoal claims. They are tactic-based provers[6] and variation of logic is typically variation in the choice of primitive inference rules made available to the proof engines. The Nuprl system also provides a uniform operator definition facility and methods for automatically "extracting" witnesses from proofs of existential claims.

A standard semantics for a computational type theoretic language is given in [2]. This semantics is often used to justify Nuprl inference rules, by arguing that the conclusion of any instance of a rule is a true sequent if the premises are. This semantics is based upon informal explanations of type theory given by Martin-Löf in [11]. The semantics is given in layers:

- A uniform syntax of expressions is stipulated. This syntax is independent of types and the expressions are used to denote types as well as their members.
- An effective partial function is stipulated as the "evaluation" relation, "expression b evaluates to expression c." It must be idempotent, that is, if b evaluates to c then c evaluates to itself. One convenient constraint is that only closed expressions (no free variables) evaluate to anything. A semantically important relation between closed expressions is what we will call Kleene-equality; two expressions are Kleene-equal when, if either evaluates to an expression, the other evaluates to the same expression.
- A class of type expressions is defined, as a subclass of the basic uniform syntax, stipulating:
  - when an expression is a type expression; we also call the expression itself a type.
  - when an expression is taken to denote a member of that type; we also say that such an expression is a member of the type, which is normally not confusing.

One constraint is that a member of a type must be closed, that is, have no free variables; another is that only closed expressions denote types.

– when two such expressions are taken to denote the same member of that type; we also sometimes call expressions of a type with a common denotation "equal" in the type. One constraint on this stipulation is that a member of a type is equal to any other expression to which it is Kleene-equal.

These are normally polymorphic type systems in that an expression may denote members in different types, and two expressions might denote the same member of one type but distinct members of another type.

- A transitive, symmetric relation on type expressions is stipulated as "intensional equality"; intensionally equal type expressions must denote the same extensional types, that is, they must have the same member expressions stipulated and the same equality relations between members stipulated. An intensional type is a type expression intensionally equal to itself. Similar to equality in a type, a type expression must be intensionally equal to any expression to which it is Kleene-equal.

- A syntax of sequents is given and a basic truth predicate True(σ) is defined for it, which is then extended to incorporate a body of operator definitions.

Inference rules are then stipulated, which amounts to defining what counts as an instance of each rule, and a rule is then justified by arguing that the conclusion sequent of each instance is true if its premise sequents are. One special feature of the Nuprl system is the method for extracting witnesses from proofs with existential import. As will be seen, this can be assimilated to the mechanism for operator definition, and the effect of implementing rules for Atoms upon extraction will be discussed below once the effect on operator definition has been.

Various different semantics for Nuprl sequents have been used for various purposes, as have some variations on their syntax.[2, 12, 8, 10] Set-theoretic semantics radically different from the standard used here have also been developed for the polymorphic type expressions normally used in Nuprl logics.[8][1]

One can view a logic as an abstraction from various semantics that validate it; then various extensions will narrow the abstraction. These semantics for Nuprl have all tended to justify a large shared body of inference rules while validating or invalidating various other characteristic rules.

A Semantics for Atoms

Here we give a semantics to sequents mentioning the Atom type expression. We will follow the form of layered semantics described above indicating how to adapt it in order to accommodate the new semantics for Atom. We ignore the uniform operator definition facility until the next section.

[1] Watch for future publication of Evan Moran's research developing this line.

Expression Syntax

The key aspect of expressions, for our purpose, is that they are discrete structures with embedded values. An expression in Nuprl, sometimes called a *term*, is a tree. Each node is labeled with of some indication of which variables become bound in which subexpression places. More importantly, for our purpose, each node is also labeled with a sequence of "value-injections," which serve to distinguish one operator from another or to inject values such as numbers or strings directly into the class of expressions, which is how we will embed Atoms. A value injection consists of a pair $v\!:\!k$ where $k \in K$ and $v \in F(k)$, for a class K of "injection-kind-indicators," and a function $F(x)$ assigning a class of injectable values to $x \in K$, K and F being parameters of the uniform syntax. When K and all $F(x)$ are discrete, i.e., have effectively decidable identity relations, then the class of expressions will also be discrete.

A simple example of a term is the canonical form of expression for non-negative integers in Nuprl, which is an expression natural$\{i\!:\!\mathrm{nat}\}$ with no immediate subexpressions, labeled with the value injections "natural$\,:\!\mathrm{tok}$, $i\!:\!\mathrm{nat}$" where $F(\mathrm{tok})$ is character strings and $F(\mathrm{nat})$ is $\mathbf{N}$.

Different values of K and F have been used over time with Nuprl logics. For our treatment of Atoms, we shall stipulate that $\mathrm{ut} \in K$ (mnemonic for *unhideable token* for reasons to be seen) be an injection-kind-indicator of the syntax, but rather than assigning a fixed class of values to ut, we leave that as a parameter. For our purpose, we can parameterize the expressions of Nuprl as $\mathrm{Term}(D)$ where D is the class $F(\mathrm{ut})$ of values x that can be injected into expressions as labels $x\!:\!\mathrm{ut}$ on expression nodes. We shall assume that K is discrete and that $F(x)$ is discrete for x other than ut, and so $\mathrm{Term}(D)$ is discrete if D is.

Our semantics for truth of sequents will quantify over possible choices for this parameter. Here are some concepts used for the semantics below. Let $\mathrm{UT}_D(t)$ be the members of D that occur as unhideable token injections somewhere in $t \in \mathrm{Term}(D)$. Notice that if $t \in \mathrm{Term}(D)$ then $t \in \mathrm{Term}(\mathrm{UT}_D(t))$ and $\mathrm{UT}_D(t)$ is $\mathrm{UT}_{\mathrm{UT}_D(t)}(t)$. When D is discrete, let $\mathrm{Replace}_D(t,f) \in \mathrm{Term}(D')$, for $t \in \mathrm{Term}(D)$ and $f \in \mathrm{UT}_D(t) \to D'$, be the result of replacing throughout t every occurrence of $i : \mathrm{ut}$ by $f(i) : \mathrm{ut}$ for all $i \in \mathrm{UT}_D(t)$. Notice that $\mathrm{Replace}_D(t,f)$ is $\mathrm{Replace}_{\mathrm{UT}_D(t)}(t,f)$.

Computation – abstract unhideable tokens

Computation is parameterized as "X evaluates$_D$ to Y" ($X \mapsto_D Y$ for short) for $X, Y \in \mathrm{Term}(D)$. We shall constrain stipulations of computation thus:

- If $b \mapsto_D c$ then $\mathrm{FV}_{\mathrm{string}}(c) \subseteq \mathrm{FV}_{\mathrm{string}}(b)$ and $\mathrm{UT}_D(c) \subseteq \mathrm{UT}_D(b)$.

  That is, free variables and unhideable tokens cannot be generated by computation, only inherited from the original data.

- If $f \in \mathrm{UT}_D(b) \xrightarrow{1-1} D'$ is an injective function, and D, D' are discrete, then $b \mapsto_D c$ iff $\mathrm{Replace}_D(b,f) \mapsto_{D'} \mathrm{Replace}_D(c,f)$.

  That is, unhideable tokens are treated abstractly; whatever concrete discrete type

is used for unhideable tokens, the computation is essentially the same.

Further, anticipating the stipulation of the type Atom and computing on its members, we further require that that

- $\text{token}\{i\!:\!\text{ut}\} \mapsto_D \text{token}\{i\!:\!\text{ut}\}$

  This will be our canonical injection of unhideable tokens into the language.

- $\text{uteq}(a,b,c,d) \mapsto_D e$ iff c, d are closed and for some $i, j \in D$,
  $a \mapsto_D \text{token}\{i\!:\!\text{ut}\}$, and
  $b \mapsto_D \text{token}\{j\!:\!\text{ut}\}$, and
  if i is j then $c \mapsto_D e$, and
  if i isn't j then $d \mapsto_D e$.

  That is, we have a form for deciding equality on discrete D.

- $\text{Atom} \mapsto_D \text{Atom}$

  This will be used in our language to denote our canonical type of unhideable tokens.

Stipulating the Atom Type

One form of type system is a relation "T denotes ϕ" construed as meaning that T is a type expression and $b\phi c$ iff b and c denote the same value in that type. This form is especially convenient for defining a type system recursively, because the usual way of forming types from prior types can be described[1, 3] as a monotonic operation on relations of this form, hence making the definition easily understandable as a least fixed-point, namely the strongest relation closed under the monotonic type-forming operator.

Then T is a type expression when T denotes some ϕ. An expression of type T is one related by ϕ to itself, and two expressions of type T are equal when they are related by ϕ. Such a relation properly describes a system of computational type theory only when it defines a partial function mapping closed expressions to two-placed symmetric, transitive relations on expressions, and these relations hold only between by closed expressions, and they respect Kleene-equality. Similarly, as stipulated earlier, if T denotes ϕ, and T and T' are Kleene-equal, then T' denotes ϕ too.

In practice, respect for relations other than Kleene-equality may also be required for particular logics, such as change-of-bound-variables or certain congruences[7].

Because defining types in computational type theory depends on an expression syntax and computation system, we parameterize as "T denotes$_D$ ϕ" the type system to capture the dependency on which values are used as unhideable tokens in Term(D). We stipulate that

Atom denotes$_D$ ϕ iff for all $b, c \in \text{Term}(D)$,
$b\phi c$ iff for some $i \in D$,
$b \mapsto_D \text{token}\{i\!:\!\text{ut}\}$ and $c \mapsto_D \text{token}\{i\!:\!\text{ut}\}$

That is, an expression of type Atom simply denotes the token (whose injection) it evaluates to. Intensional equality between types will be stipulated trivially – a type is intensionally equal to Atom iff it evaluates to Atom.

Supervaluation Semantics for Truth of Sequents

We assume that the prior basic semantics $\mathrm{True}(\sigma)$ for sequents is determined by the syntax, computation, and type system stipulations, which means it depends upon values used for unhideable tokens; making this parameter explicit, we provide a supervaluation[5] semantics of sequents in terms of $\mathrm{True}_D(\sigma)$, quantifying over choices for D. We have assimilated sequent syntax to the Nuprl uniform syntax for our convenience; indeed, it is easy in practice as well. Further, although we need to quantify over various classes that could be used as unhideable tokens, it turns out that the supervaluation semantics can be defined on expressions using strings as the tokens. Hence, for a sequent $\sigma \in \mathrm{Term}(\mathrm{string})$, let

$$\begin{array}{l}\mathrm{True}_+(\sigma) \text{ iff for some } k \in \mathbf{N},\\ \qquad\qquad \text{for every discrete class } D \text{ having at least } k \text{ members},\\ \qquad\qquad \text{for every injective function } f \in \mathrm{UT}_{\mathrm{string}}(\sigma) \xrightarrow{1-1} D,\\ \qquad\qquad \mathrm{True}_D(\mathrm{Replace}_{\mathrm{string}}(\sigma, f))\end{array}$$

Consequently, assuming standard types have been stipulated along with some standard or obvious definitions and abbreviations, and using $\vdash \Phi$ to represent a simple sequent built from a type-theoretic formula Φ, to represent the claim that Φ is true,

- for every character string x, $\mathrm{True}_+(\vdash \mathrm{token}\{x\!:\!\mathrm{ut}\} \in \mathrm{Atom})$,
- for every k, if $\mathrm{True}_+(\vdash k \in \mathbf{N})$ then $\mathrm{True}_+(\vdash \exists \in \{1..k\} \xrightarrow{1-1} \mathrm{Atom})$.

  A proof of this sequent could be witnessed in a Nuprl proof by giving a function that selects from a long enough list of distinct atom constants:

  $$\lambda\, i.\mathrm{select}(i, [\mathrm{token}\{\mathrm{abc}\!:\!\mathrm{ut}\}, \mathrm{token}\{\mathrm{def}\!:\!\mathrm{ut}\}, ...])$$

Excluding Some Rules as Invalid

On the other hand, there are formulas Φ that, while true for many D under the prior semantics, i.e., $\mathrm{True}_D(\vdash \Phi)$, do *not* satisfy the supervaluating semantics, i.e., *not* $\mathrm{True}_+(\vdash \Phi)$. For example,

- $\mathrm{True}_{\mathrm{string}}(\vdash \exists \in \mathbf{N} \xrightarrow{1-1} \mathrm{Atom})$ but *not* $\mathrm{True}_+(\vdash \exists \in \mathbf{N} \xrightarrow{1-1} \mathrm{Atom})$,

  since every finite D falsifies $\mathrm{True}_D(\vdash \exists \in \mathbf{N} \xrightarrow{1-1} \mathrm{Atom})$.
- Letting D be finite, say {"abc","def"},

  $\mathrm{True}_D(\vdash \neg\exists \in \mathbf{N} \xrightarrow{1-1} \mathrm{Atom})$ but *not* $\mathrm{True}_+(\vdash \neg\exists \in \mathbf{N} \xrightarrow{1-1} \mathrm{Atom})$,

  since not $\mathrm{True}_{\mathrm{string}}(\vdash \neg\exists \in \mathbf{N} \xrightarrow{1-1} \mathrm{Atom})$.

Thus, inference rules that would allow the proof of these formulas cannot be justified with this semantics.

The effects considered so far have been limitations on proof. If these were the only effects, then this semantics might be considered of value only as an academic explanation, practical purposes being just as well met by simply using character strings or even numbers with handy notational abbreviations. This complication of the semantics might not be worth the trouble – if we used strings or numbers instead of atoms, or added rules for exploding and imploding or enumerating atoms, so what?

However, our semantics also has productive consequences; there are inferences validated by our abstract treatment of Atoms that would be invalid for enumerable types.

The Permutation Rule

For any permutation on strings $p \in \text{string} \xrightarrow{1-1} \text{string}$, and coercing functions to functions on subdomains, and abbreviating $\text{Replace}_{\text{string}}(\sigma, p)$ by σ^p,

$\text{True}_+(\sigma)$ iff $\text{True}_+(\sigma^p)$. **[P]**

Proof:

It is enough to show that $\text{True}_+(\sigma^p)$ if $\text{True}_+(\sigma)$, since σ is $(\sigma^p)^{p^{-1}}$.

This reduces to showing that for $k \in \mathbf{N}$ and discrete D with at least k members, if $[\forall f\colon \text{string} \xrightarrow{1-1} D.\ \text{True}_D(\sigma^f)]$ then $[\forall g\colon \text{string} \xrightarrow{1-1} D.\ \text{True}_D(\sigma^{pg})]$.

Use $g \circ p$ for f, since σ^{pg} is $\sigma^{g \circ p}$.

This could be used to justify inferences from any sequent to any conclusion gotten by permuting the unhideable token injections throughout the premise.

Operator Definition – unhideable tokens

The semantics we want to use for Nuprl logics must also incorporate operator definitions. Inference rules of Nuprl are designed to remain valid whatever new operator definitions may be added, and *ordinary* rules are designed so that what counts as an instance of a rule is independent of what definitions are in force; rules designed explicitly to rewrite expressions according to the definitions in force are few.

Clearly, because substitution is so important in reasoning, being able to recognize instances of a rule independently of any operator definitions precludes the use of operator definitions that expand to produce variables not in the original expression. Nuprl operator definitions are, therefore, prohibited from having variables on the right-hand side (definiens) that are not on the left-hand side (definiendum).[2]

Rather than explain the detail of how operator definitions are stipulated and expanded, and when a collection of operator definitions is legitimate, we shall abstract to

[2]Nuprl makes other similar constraints that are not worth considering here.

the relation of two expressions having "equivalent" full expansions. We intend that the result of fully expanding an expression according to a body of definitions should leave no further expansion possible. However, we do not require full expansion to produce expression considered purely primitive in the logical sense; we allow for full expansion to result in non-primitives which could be further expanded if a larger body of definitions were used.

The ordinary process of developing bodies of operator definitions in Nuprl is to stipulate some primitives, then successively add operator definitions that cannot conflict with the primitives or already stipulated operator definitions. It is not necessary that at each stage all operators can be eliminated in favor of primitives – one can wait until an inference requiring rewriting by a definition is needed, then add an appropriate definition. Thus, not-yet-defined non-primitive operators can serve as abstractions from possible ways of defining them until a commitment to further refinement is needed.

For our purposes, we can abstractly formulate full expansion by stipulating a criterion $\pi(b)$ for primitives, which is satisfied just by instances of primitive operators, and a relation $b\,\delta\,c$ between terms, intended to represent two terms having "equivalent" full expansions, but we will not need to say generally what full expansion is. Reasonable choices for equivalent fully primitive terms would be identity or change of bound variables; stipulating operator expansion in a definition facility such as Nuprl's, in which binding operators can be defined, is simplified if any change of bound variables is allowed in stipulation of operator expansions. We do expect that if y fully expands to x then $x\,\delta\,y$. For convenience, let us say that, for a property of terms, such as π,

$\pi^\star(b)$ iff b and all its subterms satisfy π.

Let us use $\mathrm{FE}^\pi(\delta)$ to stand for a criterion of adequacy for such a representation of full expansion, for relation δ defined on Term(string). We shall assume that if $\mathrm{FE}^\pi(\delta)$ then

- δ is an equivalence relation on Term(string), **[E]**

and for any b and c such that $\pi^\star(b)$ and $b\,\delta\,c$,

- $\mathrm{FV}_{\mathrm{string}}(b) \subseteq \mathrm{FV}_{\mathrm{string}}(c)$ and $\mathrm{UT}_{\mathrm{string}}(b) \subseteq \mathrm{UT}_{\mathrm{string}}(c)$,

and for any $f \in \mathrm{UT}_{\mathrm{string}}(c) \xrightarrow{1-1} \mathrm{string}$,

- $\pi^\star(b^f)$ and $b^f\,\delta\,c^f$. **[D]**

Thus, we prohibit free variables and unhideable tokens being generated by full expansion to primitives, i.e., they cannot be hidden in definitions, and we require full expansion to primitives to treat Atom values abstractly, as well as requiring primitiveness to be independent of which Atom values may occur in a term.

Now we consider when one function for fully expanding defined operators is considered an extension of another. Let

$\delta'\,\mathrm{ext}^\pi\,\delta$ iff if $\mathrm{FE}^\pi(\delta)$ then $\mathrm{FE}^\pi(\delta')$, and
$\delta \subseteq \delta'$, and, for any $b, c \in$ Term(string),
if $\pi^\star(b)$, $\pi^\star(c)$, and $b\,\delta'\,c$ then $b\,\delta\,c$.

So extending definitions simply enlarges what terms are considered equivalent, but not which pairs of fully primitive terms are considered equivalent.

Nuprl primitive rules can be about defined operators, thus, the validity of a Nuprl rule is relative to both which operators are considered primitive and which definitions are required to be in force for future rule applications. Let us say "δ fully expands$^{\pi}$ c" just when c expands to a fully primitive term, i.e.,

δ fully expands$^{\pi}$ c iff for some b, $\pi^{\star}(b)$ and $b\,\delta\,c$.

Truth (supervaluational) relative to full expansion is defined as:

$\mathrm{True}_{+}^{\pi,\delta}(\sigma)$ iff there is a σ' such that $\pi^{\star}(\sigma')$ and $\sigma'\delta\,\sigma$,
and for all such σ', $\mathrm{True}_{+}(\sigma')$.

Hence, True sequents fully expand to primitives. We shall also require that any truth predicate to be used with the definition system respect equivalence on fully primitive terms (probably identity or change-of-bound-variables), i.e.,

$\mathrm{True}_{+}(\sigma')$ if $\mathrm{True}_{+}(\sigma)$, $\pi^{\star}(\sigma')$, $\pi^{\star}(\sigma)$, $\sigma'\delta\,\sigma$, and $\mathrm{FE}^{\pi}(\delta)$. **[T]**

The standard form of validity used to justify Nuprl inference rules involving operator definitions is that for any instance of the rule, whatever new operators may be defined, if conclusion and premises fully expand to primitive operators and the expanded premises are all true then so is the expanded conclusion. This is adapted, relative to π and δ, for the semantics of Atom by quantifying over extensions of δ:

for any n, and any rule instance with conclusion σ_0 and premises $\sigma_1 ... \sigma_n$,
and any $\delta' \in \mathrm{Term(string)} \to \mathrm{Term(string)}$,
if $\delta'\,\mathrm{ext}^{\pi}\,\delta$, and δ' fully expands$^{\pi}\sigma_0$, and $\mathrm{True}_{+}^{\pi,\delta'}(\sigma_i)$ for $i \in 1..n$
then $\mathrm{True}_{+}^{\pi,\delta'}(\sigma_0)$

The relevant fact about permutation under definitions is that if $\mathrm{FE}^{\pi}(\delta)$ then for any permutation $p \in \mathrm{string} \xrightarrow{1-1} \mathrm{string}$,

$\mathrm{True}_{+}^{\pi,\delta}(\sigma)$ iff $\mathrm{True}_{+}^{\pi,\delta}(\sigma^{p})$.

Proof:

It is enough to show that $\mathrm{True}_{+}^{\pi,\delta}(\sigma^{p})$ if $\mathrm{True}_{+}^{\pi,\delta}(\sigma)$, since σ is $(\sigma^{p})^{p^{-1}}$.
From $\mathrm{True}_{+}^{\pi,\delta}(\sigma)$, it follows by definition that there is some σ' such that $\pi^{\star}(\sigma')$ and $\sigma'\delta\,\sigma$, and $\mathrm{True}_{+}(\sigma')$, and therefore, from our basic fact about permutation **[P]**, $\mathrm{True}_{+}(\sigma'^{p})$.

From **[D]** it follows that $\pi^{\star}(\sigma'^{p})$ and $\sigma'^{p}\delta\,\sigma^{p}$, so, to show $\mathrm{True}_{+}^{\pi,\delta}(\sigma^{p})$, it is enough, by definition, to show that, for all σ'' such that $\pi^{\star}(\sigma'')$ and $\sigma''\delta\,\sigma^{p}$, $\mathrm{True}_{+}(\sigma'')$.

Since $\mathrm{FE}^{\pi}(\delta)$, δ is symmetric and transitive **[E]**, hence $\sigma''\delta\,\sigma'^{p}$.

$\mathrm{True}_{+}(\sigma'')$ then follows from **[T]**, i.e., from our global assumption that truth respects δ-equivalence on primitives.

To incorporate an unhideable token permutation rule into Nuprl logics, modify the definition system to forbid the occurrence of unhideable token injections on the right-hand side (definiens) unless they occur on the left-hand side (definiendum). Then add a rule which, applied top-down, takes a specification of a finite permutation of strings and permutes the unhideable tokens throughout the goal sequent to produce the subgoal sequent.

Witness Extraction – unhideable tokens

Nuprl logics are typically used with an *extraction* function[4], which takes a proof that some type has a member, perhaps under assumptions or depending upon some declared variables, and produces an expression for such a member. Such an extraction function would normally be specified by saying, for each rule of inference, for a proof whose top inference is an instance of the rule, how to construct a witnessing expression given witnessing expressions for the premises. Such witnesses can be quite large, so an abbreviatory mechanism is needed in the logic in order to make it practical to refer to such witnesses in other definitions and theorems.

The uniform definition facility is brought to bear on this problem by interpreting each completed, named proof of a theorem as a tacit definition whose right-hand side (definiens) is the possibly-large term extracted by composing the witness constructions stipulated for the inference rules. The left-hand side (definiendum) of this tacit definition is an expression derived from the name of the proof and may involve some other value-injections.

Thus, incorporating the permutation rule into Nuprl requires making sure the tacit definitions involved conform the the constraints on full expansion we have already discussed, especially, one must not extract any unhideable tokens not mentioned in the left-hand side (definiendum). One strategy is to build the left-hand side before the full proof is built, prescribing *a priori* which unhideable tokens may be extracted – this will limit the application of inference rules when the proof is developed. Another strategy is to generate the left-hand side only after the proof is sufficiently developed to determine all possible unhideable tokens that might be extracted. Both strategies can co-exist, entailing some complexity in the logic.

Summary

We have provided a supervaluating semantics for treating Atoms abstractly in Computational Type Theory, specifically for Nuprl logics. It provides a principled explanation for desirable gaps in provability without positing novel kinds of entities, nor relying in any way upon constructivity of the logic. Beyond that, though, we have justified a rule that allows inference by renaming Atom values, and explored the impact of introducing this new rule upon notational definition as used in the logics. We have *not* reported here upon how these methods were actually implemented in the Nuprl system – we have simply provided the semantic basis.

References

[1] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In D. Gries, editor, *Proceedings of the 2^{nd} IEEE Symposium on Logic in Computer Science*, pages 215–224. IEEE Computer Society Press, 1987.

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

[3] Robert L. Constable. Types in logic, mathematics and programming. In S. R. Buss, editor, *Handbook of Proof Theory*, chapter 10. Elsevier Science, 1998.

[4] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[5] Kit Fine. Vagueness, truth and logic. *Synthese*, 30:265–300, 1975. Reprinted in Keefe and Smith[9].

[6] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[7] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4^{th} IEEE Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society Press, 1989.

[8] Douglas J. Howe. Semantics foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101, pages 85–101, Berlin, 1996. Springer-Verlag.

[9] Rosanna Keefe and Peter Smith, editors. *Vagueness: A reader*. MIT Press, Cambridge, MA, 1997.

[10] Alexei Pavlovich Kopylov. *Type Theoretical Foundations for Data Structures, Classes, and Objects*. PhD thesis, Cornell University, Ithaca, NY, 2004.

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

[12] P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.