Importing Mathematics from HOL into Nuprl

Douglas J. Howe

Bell Labs, Lucent Technologies
700 Mountain Ave., Room 2B-438
Murray Hill, NJ 07974, USA.
howe@bell-labs.com

Abstract. Nuprl and HOL are both tactic-based interactive theorem provers for higher-order logic, and both have been used in many substantial applications over the last decade. However, the HOL community has accumulated a much larger collection of formalized mathematics of the kind useful for hardware and software verification. This collection would be of great benefit in applying Nuprl to verification problems of real practical interest. This paper describes a connection we have implemented between HOL and Nuprl that gives Nuprl effective access to mathematics formalized in HOL. In designing this connection, we had to overcome a number of problems related to differences in the logics, logical infrastructures and stylistic conventions of Nuprl and HOL.

1 Introduction

Nuprl [2] and HOL [4] are general-purpose interactive theorem proving systems with a number of similarities: their logics are higher-order type theories, their automated reasoning facilities are based on the tactic mechanism of LCF [3], and their main application has been to formal reasoning about computation. Both systems have been the focus of a great deal of research over the last decade. However, the overall thrust of the two research communities has been rather different.

Much of the work of the Nuprl project has involved the core of the system. There has been work on Nuprl's constructive type theory, on the proof editors, and on the basic architecture of Nuprl's automated reasoning support. In contrast, the core of the HOL system has remained stable for many years. The system has attracted a large number of users, and a great deal of the effort in the HOL community has gone into building the libraries of formal mathematics that are needed for verifying hardware and software of practical interest. There have been a number of substantial applications of Nuprl (see [7] for a recent example), but there has been nothing like the HOL community's sustained effort to formalize mathematics useful for verification.

The mathematics formalized in HOL would be of great benefit in applying Nuprl to verification problems of real practical interest. The goal of the present work is to connect the two systems so that mathematics can be imported from HOL into Nuprl.

Because of the similarities of the two systems, one might think that this is an easy thing to do. In one sense, it is. Just about any theorem-prover can embed the logic of any other simply by formalizing the syntax of proofs. However, the goal here is a practical one: to be able to effectively use HOL mathematics in Nuprl proofs. Not just any embedding will do. We need a strong connection between the mathematics developed in Nuprl and the mathematics imported from HOL, so that HOL facts will be applicable in Nuprl proofs. Furthermore, Nuprl's automated reasoning programs must be able to incorporate the HOL mathematics.

It turns out, perhaps surprisingly, that we have had to deal with a number of substantial problems. Some of the difficulties stem from the fact that the HOL and Nuprl type theories are, in fact, very different in some practically important ways. Nuprl has a constructive type theory, based on a type theory of Martin-Löf[9]. The theory contains an untyped programming language, and all objects have a computational interpretation. Programs are reasoned about directly in the logic, and the constructivity of the theory means that programs can be synthesised from proofs. On the other hand, HOL's theory is classical, and the way mathematics is encoded is similar to the way ordinary mathematics is done in ZF set theory. Functions are built in, but other objects, such as integers and lists, are given set-theory-like encodings with the aid of the "select operator" $@x \in T.\ P(x)$, which denotes some x of type T such that $P(x)$.

Aside from the logics, there are other differences between HOL and Nuprl that cause difficulties for importing HOL mathematics. In particular, there are substantial differences in:

- the logical infrastructure, including the definition and theory mechanisms,
- tactics,
- stylistic conventions for writing definitions and theorems, and
- implementation languages (HOL90 is SML, Nuprl is Common Lisp and "classic" ML).

Instead of reconciling the two logics, why not just replace Nuprl's type theory by HOL's? One answer is that Nuprl's type theory offers a number of advantages over HOL's. These include the following.

- *Expressive power of the type system.* Nuprl has subtypes and dependent function types. Also, through the use of universes and "sigma" types, one can express modules similar to the kind found in Standard ML [11].
- *Constructivity.* Experience with Nuprl has shown that for the mathematics of programs, constructivity comes at essentially no cost. Proofs about computationally meaningful objects such as hardware and software are naturally constructive, or can be made so with little effort. Thus one can, for example, build the same kinds of proofs as one does in HOL, with the additional benefit that programs can be extracted from the proofs.
- *Writing programs.* Nuprl includes a programming language which, while primitive, includes many of the features, such as function definition by general recursion, of a conventional functional programming language.

Most of these features have been advocated, in some form, as extensions to HOL. See, for example, [10, 8, 13]. Also, some of the type-theoretic features of Nuprl have been adopted in the PVS system [12]. Nuprl also has a large amount of automated support for making effective use of these features. Of course, one of the constraints on the embedding of HOL is that use of HOL mathematics does not prevent us from taking advantage of these features.

Another advantage of Nuprl is its user interface. Because of its separation of display and abstract syntax, Nuprl allows a great deal of flexibility in the use of notations in both the presentation and editing of mathematical syntax. A key component is Nuprl's novel structure editor. For details, see [6].

The main motivation for this work has been to make Nuprl more effective. However, another way of viewing the work is as the first case study of cooperation and result-sharing between different interactive theorem provers.

The body of the paper is divided into two parts. The first part gives a simple example illustrating the connection we have implemented between Nuprl and HOL90. The second part is a more detailed discussion of the problems we encountered and how we solved them. The hardest problem was one of semantics. To accommodate the embedding, we extended the Nuprl semantics to incorporate an operator similar to HOL's select. This extension is the subject of another paper [5], and will only be described briefly here. The other problems, though numerous, were not so difficult to deal with. This section also discusses some problems which we have not solved.

The work described in this paper is somewhat "in progress". The basic connection between the systems has been implemented, but it has not yet been tested in any substantial practical examples. We are currently working on a verification the SCI cache-coherency protocol [1] based on an importation of the HOL90 theory "List" (and all of its ancestors). A report on this work will be available soon via *http://www.research.att.com/howe*.

2 An Example

An HOL theory consists of some type and individual constants, some axioms, usually definitional, constraining the constants, and a set of theorems following from the axioms (and the axioms of ancestor theories). In contrast, the Nuprl logic is fixed; new constants and axioms are not introduced when building Nuprl's analogue of HOL theories. Formal mathematics in Nuprl is organized as a single list of *abstractions* and theorems. An example of an abstraction is

$$abs\_val(x) \;==\; if\ x < 0\ then\ -x\ else\ x$$

which defines a new operator *abs_val* that takes one argument. This can be thought of as a one-argument term-macro. Abstractions in Nuprl are extra-logical. The meaning of an expression containing abstractions is defined to be the meaning of the expressions obtained by expanding all abstractions (*i.e.* repeatedly replacing all occurrences by the corresponding right-hand sides). Nuprl

also lets us define *display forms* for abstractions. For example, if we make the display-form definition

$$|\langle x \rangle| \;==\; abs\_val(\langle x \rangle)$$

then any subsequent occurrence of an expression of the form $abs\_val(e)$ will appear to the user as $|e|$. Because Nuprl's editors are structural, parsing/unparsing issues do not arise.

Consider now a truncated version of the HOL theory "bool" which introduces, among other things, some of the usual connectives of higher-order logic.

```
Parents:     min

Types:

Constants:   ?    :('a -> bool) -> bool
             T    :bool
             !    :('a -> bool) -> bool
             /\   :bool -> bool -> bool
             \/   :bool -> bool -> bool
             F    :bool
             [...]

Axioms:      BOOL_CASES_AX  |- !t. (t = T) \/ (t = F)
             IMP_ANTISYM_AX  |- !t1 t2. (t1 ==> t2) ==> (t2 ==> t1)
                                        ==> (t1 = t2)
             [...]

Definitions: EXISTS_DEF  |- $? = (\P. P ($@ P))
             TRUTH  |- T = (\x. x) = (\x. x)
             FORALL_DEF  |- $! = (\P. P = (\x. T))
             AND_DEF  |- $/\ = (\t1 t2. !t. (t1 ==> t2 ==> t) ==> t)
             OR_DEF  |- $\/ = (\t1 t2. !t. (t1 ==> t) ==> (t2 ==> t)
                                           ==> t)
            [...]

Theorems:
```

The meaning of this theory is that for all values of the constants declared in the `Constants` section of the theory (and all ancestor theories), if the values satisfy the formulas in the `Axioms` and `Definitions` section of the theory (and all ancestor theories), then the formulas in the `Theorems` section are all true.

Suppose that we want to import this theory into Nuprl. We first invoke a program in HOL90 that writes the theory to a file in an intermediate form more suitable for Nuprl. This program also takes into account some hints supplied by the user. For example, for the theory `bool` some new names were supplied for some of the constants (most necessary renaming is done automatically). We then invoke a program in Nuprl that translates this file into the following Nuprl library fragment.

```
*C bool_begin   ************ BOOL ************
```

```
*A h_exists     ∃z_0:z_1. P1[z_0] ==  []
*A T            T ==  []
*A h_all        ∀z_0:z_1. P1[z_0] ==  []
*A h_and        P1 ∧ P2 ==  []
*A h_or         P1 ∨ P2 ==  []
*A F            F ==  []
[...]
#T h_exists_wf  ∀'a:S. ∀z_1:'a → o.  (∃z_0:'a. z_1[z_0]) ∈ o
[...]
#T ...   ∀'a:S. (λz_0. ∃z_1:'a. z_0 z_1)
                       = (λP. P(@z_0:'a. P z_0))
#T ...   [T] ⟺ (λx. x) = (λx.x)
#T ...   ∀'a:S. (λz_0. ∀z_1:'a. z_0 z_1) = (λP. P = (λx. T))
#T ...   (λz_0,z_1. z_0 ∧ z_1)
         = (λt1,t2. ∀t:o. (t1 ⇒ t2 ⇒ t) ⇒ t)
#T ...   (λz_0,z_1. z_0 ∨ z_1)
         = (λt1,t2. ∀t:o. (t1 ⇒ t) ⇒ (t2 ⇒ t) ⇒ t)
#T ...   [F] ⟺ (∀t:o. [t])
```

Some of the object names have been elided for the sake of compactness. Also, Nuprl's type of booleans has been given the display form `o`.

Each line in the library fragment describes a single object. The second character in each line gives the kind of object: `C` for comment, `A` for abstraction and `T` for theorem. The first character is the status: `*` for complete and `#` for incomplete (e.g. when a theorem's proof has not been completed).

There are a number of apparent differences between this fragment and the HOL theory. We point out some of these here, but defer the explanations to the next section.

For each constant in the HOL theory, there are two Nuprl objects. The first is an abstraction with a right hand side which is initially a fixed constant. Corresponding to the constant `/\` we have the abstraction named `h_and`. The left-hand side of the abstraction definition is actually `h_and(P1;P2)`. The system displays it as `P1 ∧ P2` because it is using a display form we have associated with the operator `h_and`. A Nuprl library may contain any number of such user-created display-form definitions. They have no logical significance; their only relevance is in display and editing of mathematical text.

Corresponding to the HOL existential operator `?` is the abstraction `h_exists`. The left-hand side here is actually `h_exists(z_1; z_0. P1[z_0])`. This is a *second-order* abstraction: `z_1` ranges over terms, but `P1` ranges over terms with a distinguished free variable, and the expression `P1[z_0]` stands for the substitution of `z_0` for the distinguished variable. As an example of the use of this operator, to express that there exists a natural number n equal to 0 one would write `h_exists(N; n. n=0)`. On the right-hand side of the definition of `h_exists` one may apply $P1$ to any term. Note that the operator here takes two arguments, while the HOL constant takes only one.

The second object associated with a HOL constant is a *well-formedness theorem*, initially unproven. This essentially asserts that the abstraction has "same" type as the HOL constant. The fragment above shows only one such theorem, for `h_exists`. Note that we must explicitly quantify over the type `'a` that is the first argument to `h_exists`. `S` represents the type of all HOL types. The expression `z_1[z_0]`, after expanding abstractions, is just the function-application of `z_1` to `z_0`.

Each definitional axiom of the HOL theory is mapped to a corresponding Nuprl theorem, initially unproven. Consider first the second theorem following the second `[...]`. This corresponds to the HOL definition `TRUTH`. Note that an HOL equality has been replaced by $\Longleftrightarrow$. In order to make translations of HOL theorems more directly applicable in Nuprl proofs, the main logical connectives of HOL have been translated into their Nuprl analogues. Thus HOL's "iff", which is equality over the type `bool`, maps to Nuprl's "iff". There is also an operator, denoted by $[\cdot]$, that coerces a value of type `bool` to a Nuprl proposition.

Note the difference between `AND_DEF` and the corresponding Nuprl theorem. "And" in HOL is just a function of two arguments. In Nuprl, it becomes a binary operator, and to make this into a function we need to explicitly abstract it.

If there were theorems in the HOL theory, then they would be translated in the same way as the axioms.

Given this library fragment, the user must now "instantiate" this Nuprl representation of an HOL theory with Nuprl objects. In particular, the user must do the following.

1. Fill in the right hand sides of the abstraction definitions. Usually this can be accomplished by cutting and pasting from the definitional axiom for the corresponding HOL constant. The key point here is that we will use computationally meaningful Nuprl objects to fill in these abstractions. For example, suppose `bool` (declared in the HOL theory "min") has been given the following definition in Nuprl as a subset of the integers.

$$\texttt{bool} == \{\texttt{x:Z} \mid \texttt{x=0} \vee \texttt{x=1}\}.$$

We can then give a computable definition for, e.g., `h_and`:

$$\texttt{P1} \wedge \texttt{P2} == \texttt{P1*P2}.$$

There is a handful of HOL functions that cannot be given computable definitions. `h_exists` is such an example.

2. Prove the well-formedness theorems. These proofs are almost always done automatically by expanding the abstraction and running Nuprl's catch-all "autotactic".

3. Prove the translations of the axioms. In the case of definitional axioms, these proofs are usually trivial. They can, as in the case of type definitions, be quite non-trivial, though.

4. "Prove" the translations of the theorems. This is done automatically by using a "tactic" which works as follows. To prove a theorem $\vdash \phi$, the tactic first rewrites all the Nuprl logical operators to their HOL analogues. This reduces proving ϕ to proving $[\psi]$ where ψ is an expression of type *bool*. Using the name of the theorem being proven, the tactic then looks up the theorem in Nuprl's internal image of the HOL theories loaded so far. It checks that the statement of this theorem is identical to ψ, then checks the completeness of the Nuprl import of the theory containing the theorem as well as the completeness of all ancestors. A Nuprl library fragment corresponding to a HOL theory is complete if all required axioms and and well-formedness theorems are present and completely proven. If the checks succeed, then Nuprl simply marks the theorem as proven.

Note that we are simply trusting HOL that its theorems are true. The correctness of our connection between HOL and Nuprl also relies on the correctness of the implementation of three programs: the SML program that preprocesses HOL theories and writes them to files, the Lisp/ML program that brings the files into Nuprl, and the ML program that marks imported facts as proven. In addition, soundness depends on the correctness of the semantic connection discussed in the next section.

3 Problems and Solutions

In this section we discuss some of the problems we encountered and how we solved them. Most of the problems relate to differences between the two type theories and the ways they are applied.

3.1 Semantics

HOL has a standard set theoretic semantics where propositions are booleans and function spaces contain all functions in a set-theoretic sense. In Nuprl, all semantic objects, including types themselves, are terms in an untyped programming language. One can think of this language as something like pure Lisp, or a variant of the untyped lambda-calculus. Thus, in the Nuprl semantics,

$$\lambda x.\ \mathit{if}\ x{=}0\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{false}$$

is a member of the type $N \rightarrow \mathit{bool}$. The reason is that if we evaluate the application of this expression to any natural number, the result is a boolean. In contrast, the corresponding semantic object in HOL is

$$\{(0, \mathit{true}), (1, \mathit{false}), (2, \mathit{false}), \ldots\}.$$

In Nuprl, a term having a type is a semantic property. Something has a function type only if it has the right input-output behaviour. Thus we have the following typing, which has no direct translation into HOL.

$$(\mathit{letrec}\ f(i) = \mathit{if}\ i{>}100\ \mathit{then}\ i{-}10\ \mathit{else}\ f(f(i{+}11))) \ \in\ N{\rightarrow}N$$

Semantically, the function has this type simply because it terminates with a natural number on any natural-number input. In Nuprl, this typing requires a non-trivial proof, similar to what one would do to show termination informally (it requires a clever choice of well-founded ordering — try it!). This is just an instance of the general fact that in Nuprl, one can reason directly about functions defined by general recursion, but not in HOL.

Because the programming language is untyped, sensible terms can have "junk" subterms, as in

$$\textit{if } 0 = 0 \textit{ then true else } 17 + \textit{"foo"}$$

which has type *bool* since it evaluates to *true*.

We have reconciled these two semantics by extending the Nuprl semantics to include set-theoretic objects of the kind found in HOL. In particular, we add a collection of set-theoretic functions as new constants in Nuprl's programming language. We need to extend the notion of evaluation in the language. For example, we need to be able to evaluate applications like $\phi(e)$ where ϕ is one of the injected set theoretic functions and e is an arbitrary term. To do this, we introduce a notion of approximation. The assertion $\alpha \lhd e$ means that the set theoretic object α approximates the term e.

The evaluation relation of Nuprl's programming language is inductively defined by a set of rules. For example, the rule for ordinary function application is

$$\frac{f \Downarrow \lambda x.\, b \quad b[a/x] \Downarrow v}{f(a) \Downarrow v}$$

We add rules for $\lhd$, and add new evaluation rules for set theoretic objects. For example, we add two rules for set-theoretic functions ϕ:

$$\frac{f \Downarrow \phi \quad (\alpha,\beta) \in \phi \quad \alpha \lhd a}{f(a) \Downarrow \beta} \qquad \frac{\forall\, (\alpha,\beta) \in \phi.\ \beta \lhd b[\alpha/x]}{\phi \lhd \lambda x.\, b}$$

We illustrate this extension of evaluation with a few examples. Let $\phi = \{(0,4),(1,5)\}$, $\phi' = \{(0,2)\}$ and $\psi = \{(\phi,17),(\phi',18)\}$. We have

- $\phi(0+0) \Downarrow 4$ because $0 \lhd 0+0$.
- $\phi \lhd \lambda x.\, x+4$, but not $\phi' \lhd \lambda x.\, x+4$.
- $\psi(\lambda x.\, x+4) \Downarrow 17$.

Another set-theoretic notion we need to deal with is quotienting. Consider the rational numbers. In set theory, the rationals are represented as a set of pairs of integers, quotiented by the appropriate equality. The quotient in set theory is the set of all equivalence classes. In HOL, equivalence classes can be represented as predicates. This kind of representation of quotients is problematic computationally. For example, how does one do computations over rational numbers if a rational is represented as a function of type $\textit{int} \times \textit{int} \rightarrow \textit{bool}$?

To deal with this, we add equivalence classes ξ to our language, and for computational purposes we also add a "polymorphic" equivalence class constructor. The polymorphic class $[\alpha]$ can be thought of as standing for any equivalence class ξ such that ξ has α as a member. We have the following evaluation/approximation rules for equivalence classes.

$$\frac{\alpha \in \xi \quad \alpha \lhd a}{\xi \lhd [a]} \qquad \frac{a \Downarrow [e] \quad f(e) \Downarrow v}{f \cdot a \Downarrow v} \qquad \frac{a \Downarrow \xi \quad \forall \alpha \in \xi.\ \beta \lhd f(\alpha)}{f \cdot a \Downarrow \beta}$$

The first rule captures the intuition given above for $[\cdot]$. The second and third rules describe how to compute with equivalence classes. To evaluate $f \cdot a$, where a evaluates to some equivalence class, we want to check that f returns the same value whenever it is applied to a member of the equivalence class, and then to return this value. When a evaluates to the polymorphic class $[e]$, there is no check to perform, so we just evaluate $f(e)$. In the case where e evaluates to a set-theoretic class ξ, we approximate the check by guessing some set-theoretic value β and checking that it approximates $f(\alpha)$ for every $\alpha \in \xi$. This introduces non-determinism.

Crucial for giving constructive implementations of HOL-defined types is Nuprl's quotient type. A simple example should suffice here. The quotient type $(x, y) : N//even(x - y)$ can be read as the quotient of N by the relation that equates numbers iff they have the same parity. Let $\xi_1 = \{0, 2, \ldots\}$ and $\xi_2 = \{1, 3, \ldots\}$. The type has as (canonical) members ξ_1, ξ_2 and $[\underline{n}]$ for $n \geq 0$. We have $\xi_1 \lhd [2]$ but not $\xi_2 \lhd [2]$. Also, if

$$f = \lambda x.\ \textit{if evenp(x) then 0 else 1}$$

then $f \cdot [2] \Downarrow 0$ and $f \cdot \xi_2 \Downarrow 1$.

This semantics also justifies introduction of an analogue of HOL's select operator. We can extend Nuprl's programming language with an evaluation rule for select as follows. Note that non-emptiness of a type is taken to represent truth of the corresponding proposition.

$$\frac{T \Downarrow \gamma \quad \forall \alpha \in \gamma.\ P[\alpha/x] \Downarrow \gamma_\alpha \quad \alpha_0 \in \gamma \textit{ of minimum rank such that } \gamma_{\alpha_0} \neq \emptyset}{@x \in T.\ P \quad \Downarrow \quad \alpha_0}$$

Showing that a sensible semantics can be built based on the ideas described above is the subject of [5]. In this semantics, the Nuprl types are exactly the programs that evaluate to some set γ. The members of such a type are the members of γ together with all terms approximated by some $\alpha \in \gamma$.

3.2 Logic

In HOL logic is given a Tarskian semantics. A proposition is either true or false. In Nuprl, all propositions are represented as types. False propositions are empty types, and true propositions are types whose members represent the "computational content" of the proposition. For example, consider the proposition

$\forall x \in N.\ \exists y \in N.\ \ x < y\ \&\ prime(y)$. In HOL, the meaning of this is just the boolean *true*. In Nuprl, it is the type of all programs taking input $x \in N$ and returning as result a prime y with $x < y$.

We can give HOL propositions a direct interpretation (a "shallow" embedding in HOL parlance). The base logic has three constants

$$
\begin{aligned}
= &: \ 'a \rightarrow\ 'a \rightarrow\ bool \\
==> &: bool \rightarrow\ bool \rightarrow\ bool \\
@ &: (\ 'a \rightarrow\ bool) \rightarrow\ 'a
\end{aligned}
$$

for equality, implication, and "select", respectively. The *'a* is a type variable. Given Nuprl's select operator, and the definition of *bool* from the previous section, it is trivial to give definitions in Nuprl for these constants, and to show that they have the required types.

This embedding by itself is not enough, however, since Nuprl's reasoning machinery is built around Nuprl's own logical operators. Fortunately, we can prove in Nuprl that the two different representations of logic are, in a sufficient sense, equivalent. For example, we can prove

$$\forall x, y \in bool.\ [x \Rightarrow y] \Leftrightarrow ([x] \Rightarrow [y])$$

where $\Rightarrow$ is overloaded notation, standing for HOL's version of implication on the left, and Nuprl's on the right, and where $[b]$ (not to be confused with the equivalence class constructor) is defined to be the proposition $b = true$. Also, we can prove

$$\forall A \in S.\ \forall P \in A \rightarrow bool.\ [\exists x \in A.\ P(x)] \Leftrightarrow (\exists x \in A.\ [P(x)]).$$

Putting a direct embedding of an HOL proposition into a form more suitable for Nuprl's tactics is thus just a matter of exhaustively applying rewrite rules such as the above.

3.3 Constructive vs Classical

In dealing with logic as we have above, we run the risk of losing the ability to extract programs from proofs. The program extracted is a member of the type representing the theorem proved. With our new semantics, we can always extract some "program", and it will have the right properties under evaluation, but the problem is that it might contain instances of the select operator, which is not computable. Nevertheless, we want to retain the ability to prove a theorem constructively and be assured that the extraction is computable.

The equivalence of the two representations of logic is highly non-constructive. In general, the object extracted from a proof of equivalence of two formulas will contain the select operator. Thus if we have tactics making unrestricted use of facts imported from HOL, it would be easy to unwittingly introduce a non-computable element.

The main reason we can solve this problem is because equalities in Nuprl have no computational content. So, for example, if a universally quantified equation is proved, then the program extracted from the proof is simply a constant function. This has two main consequences. First, if we are proving an equation (possibly under assumptions) in Nuprl, we can safely use any HOL theorem whatsoever. Second, no matter what we are proving, it is always safe to use HOL facts, such as universally quantified equations, that have no computational content. Fortunately, the vast majority of HOL theorems fit this category, and the vast majority of the work in proving any theorem about software involves computationally trivial facts (mostly equations and inequations). Most of the work in Nuprl proofs is done by term rewriting. All the programs that apply term rewriting can safely use any HOL theorem.

It is easy to modify the Nuprl system to ensure that non-computable "programs" are not inadvertently extracted from proofs. For example, we can add a bit to each proof node, where a true bit means that the extracted program of the subproof rooted at the node must not contain the select operator. The user sets the bit at the root, and the system computes the bit when the proof is extended by refinement, setting it to false when the node being refined has a conclusion which is computationally trivial, and simply propagating it otherwise. Inference steps may not mention the select operator, or use lemmas whose top bit is false, if the bit at the node being refined is true. This scheme for containing non-constructive reasoning has not yet been implemented, so it is currently up to the user to exercise appropriate care.

3.4 Constructivizing HOL Type Definitions

When new types are introduced in HOL, they are often given non-constructive implementations. Fortunately, when we translate into Nuprl we are not stuck with these implementations. This is because types in HOL are only defined up to isomorphism.

Consider the disjoint union $A + B$ of types A and B. In HOL, members of $A + B$ are represented as functions of type $bool \rightarrow A \rightarrow B \rightarrow bool$. For example, the injection of $a \in A$ into $A+B$ is represented as the function which returns true iff its first argument is *true* and its second argument is a. $A + B$ is axiomatized to be isomorphic to the collection of such representations by the following.

```
IS_SUM_REP
|- !f.
     IS_SUM_REP f =
     (?v1 v2.
       (f = (\b x y. (x = v1) /\ b)) \/
       (f = (\b x y. (y = v2) /\ ~b)))
sum_TY_DEF |- ?rep. TYPE_DEFINITION IS_SUM_REP rep
```

IS_SUM_REP is thus defined to be predicate that picks out the members of $bool \rightarrow A \rightarrow B \rightarrow bool$ that serve as representations, and the second axiom states the

existence of a bijection between $A + B$ and collection the objects satisfying IS_SUM_REP.

In contrast, Nuprl has a built-in type for disjoint union, with members $inl(a), inr(b)$. We use this as a definition of the disjoint union imported from HOL. We are then left with proof obligations to show that the above two axioms hold for this implementation, i.e. to show that Nuprl's disjoint union type and the HOL representation are in bijective correspondence. The Nuprl proof of this is non-constructive, but we will never need to refer to these axioms in other proofs.

3.5 Polymorphism and Type Inference

Although Nuprl's programming language is itself untyped, in practice the typing of programs follows a rather familiar type assignment style *á la* Curry. A difference is that types must be explicitly quantified in Nuprl. Since Nuprl has type universes, we can define a type S that contains all (small) types that are non-empty. The set S is sufficiently large to represent all HOL types. So, in HOL the polymorphic identity has the following typing

$$\lambda x.\ x \in\ 'a \rightarrow\ 'a,$$

whereas in Nuprl we would write

$$\forall\ 'a \in S.\ \lambda x.\ x \in\ 'a \rightarrow\ 'a.$$

Despite this similarity, there is still the crucial difference that type assignments can be statically determined in HOL. Given any expression, most-general types can be determined for the expression and all of its subexpressions. This is not possible in Nuprl. This gives rise to a slightly nasty problem with the select operator.

Consider the HOL expression $@x.P(x)$. Whenever this expression is used in some other expression, we can determine a type T for it, and the expression will denote some value in (the meaning of) T. In Nuprl, we cannot determine such a type in general, and the type must be passed in as an argument. Thus the expression is translated into Nuprl as $@x \in T.\ P(x)$.

Because of this, a definition whose right-hand side mentions the select operator may translate to an operator that passes type arguments. Fortunately, this happens relatively rarely. The default behaviour of the translator is to not pass type arguments. Exceptions must be indicated by the user through the "hint" mechanism.

3.6 Constants vs Operators

As indicated by the example in Section 2, definitions in Nuprl are typically operators with arguments. In HOL, defined objects that take arguments are represented as functions. This could also be done in Nuprl, by simply making each abstraction 0-ary and using λ-abstractions on the right-hand side. The reason

this is not done is partly because of the undecidability of type-checking and type-inference in Nuprl. The details are somewhat technical, but it has turned out to be easier to organize facts related to typability (and a few other properties) around operators with arguments. This approach has also allowed us to incorporate second-order pattern matching in a straightforward way in type checking and rewriting.

Whether or not this difference is valuable, Nuprl's tactic collection relies on it heavily, so to incorporate HOL facts the constants need to be adjusted accordingly. For each constant in an HOL theory to be imported, an *arity* is computed. The arity is simply the number of (curried) arguments indicated by the type of the HOL constant. The user can also supply arities for cases when this default arity is undesirable. For example, the type of the operator o for composing two functions gives a default arity of 3, while the desired value is 2. Also, occasionally one will want the constant to map to a binding operator. This is the case, for example, for a constant declared to be a "binder" in HOL, e.g. `h_exists` in Section 2.

The arities are used when translating HOL expressions to Nuprl ones. When a constant of arity n is applied to at least n arguments, the n-ary application is replaced by an instance of the corresponding n-ary operator. If there are fewer than n arguments, η-expansions are first done to add a sufficient number of arguments.

3.7 Partial Functions

In Nuprl, as in HOL, function types are total: a function of type $A \rightarrow B$ produces a value of type B for every input of type A. However, most partial functions can be given convenient types in Nuprl because of Nuprl's subset type. For example, consider *hd*, the function that takes the head of a list. In Nuprl, $hd([])$ is undefined, but we can give *hd* the type

$$\{\, l \in N\ list \mid l \neq nil \,\} \rightarrow N,$$

for example.

In HOL, *hd* is defined on all lists: $hd([])$ is $@x \in \text{'}a.\ true$. This is a problem because when the theory containing *hd* is translated into Nuprl, we have to prove the well-formedness theorem for *hd*, and this requires showing that the Nuprl-defined *hd* is defined on the empty list. This forces us into giving the uncomputable HOL definition for *hd* in Nuprl. This kind of use of @ appears to be frowned on in the HOL community and does not seem to arise much.

Nevertheless, such cases do arise. Fortunately, in all the cases we have examined so far, we can work around the problem. Consider again the example of *hd*. We make the uncomputable definition for *hd*, but we also define a computable version, call it *hd'*, and use this version in all subsequent definitions and in all theorems except for the definitional axiom for *hd*. This works because, although *hd* is defined on the empty list, this property is not taken advantage of. Consider a theorem asserting that the head of the list formed by consing x onto l is x.

The theorem that is directly justified by imported HOL theories is something like

$$\forall x \in A.\ \forall l \in A\ list.\ hd(x :: l) = x.$$

But in Nuprl this formula can be proved equal to

$$\forall x \in A.\ \forall l \in A\ list.\ hd'(x :: l) = x$$

by using the following lemma as a conditional rewrite rule:

$$\forall l \in A\ list.\ l \neq nil \Rightarrow hd(l) = hd'(l).$$

Note that this equivalence is within Nuprl, so it is irrelevant whether the HOL proof of the theorem relied on *hd* being defined on the empty list.

3.8 Theorems and Tactics

A number of tactics require additional information to make effective use of the theorems in Nuprl's library. For instance, some tactics require theorems of a certain kind to be annotated with special abstractions that have no logical significance but provide guidance to the tactic. More commonly, tactics require explicit indications, either by naming conventions or by explicit updates to reference variables via ML objects in the library, of relevant theorems and associated information. Currently, this must be dealt with by hand for imported theorems, just as with ordinary Nuprl theorems.

3.9 Unsolved Problems

One immediate problem we have not dealt with yet is HOL definitional packages. For example, there are packages that simulate various convenient forms of inductive definition and provide useful tactics for reasoning about the definition. While the theory objects generated by these packages can be readily imported using our scheme, the result is too low-level in some ways. The connection of the translated objects with the original higher-level definition is lost. It remains to be seen how effective Nuprl's tactics will be with such theories.

Other problems to be addressed in the future include abstract theories and making importation more incremental. Also, it might be interesting to try to import HOL tactics as well. The main obstacle to doing so is that HOL's tactic mechanism is incompatible with Nuprl's. In Nuprl, each time a tactic is applied by the user to refine a node in a proof tree, one is guaranteed by the system that the inference is sound. There is no such guarantee in HOL; soundness is guaranteed only for complete proofs. One way to fix this would be to redefine HOL's tactic type to be like Nuprl's, replacing the type *thm* with the type *proof* of (possibly incomplete) proofs. Making such a change would probably only affect the lowest levels of the system. Another way, incurring only a slight risk of unsoundness, is to take the approach of HOL's subgoal package.

References

1. *Part IIIA: SCI Coherence Overview, 1995.* Unapproved draft IEEE-P1596-05Nov90-doc197-iii.
2. R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
3. M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.
4. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge, UK, 1993.
5. D. J. Howe. Semantics foundations for embedding hol in nuprl. *Proceedings of AMAST'96*, 1996. (to appear).
6. P. Jackson. *Nuprl 4.2 Reference Manual.* Cornell University, 1995. Available from ftp://cs.cornell.edu/pub/nuprl/doc.
7. P. B. Jackson. Exploring abstract algebra in constructive type theory. In A. Bundy, editor, *12th Conference on Automated Deduction*, Lecture Notes in Artifical Intelligence. Springer, June 1994.
8. B. Jacobs and T. Melham. Translating dependent type theory into higher order logic. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1993.
9. P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
10. T. Melham. The HOL logic extended with quantification over type variables. *Formal Methods in System Design*, 3(1–2):7–24, August 1993.
11. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
12. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proceedings of CAV'96*, Lecture Note in Computer Science. Springer Verlag, 1996.
13. M. van der Voort. Introducing well-founded function definitions in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, volume A-20 of *IFIP Transactions*, pages 117–131. North-Holland, 1993.

This article was processed using the LaTeX macro package with LLNCS style