Hybrid Interactive Theorem Proving using Nuprl and HOL[⋆]

Amy P. Felty and Douglas J. Howe

Bell Labs, Lucent Technologies
700 Mountain Ave., Murray Hill, NJ 07974, USA
{felty,howe}@research.bell-labs.com

Abstract. In this paper we give the first example of a significant piece of formal mathematics conducted in a hybrid of two different interactive systems. We constructively prove a theorem in Nuprl, from which a program can be extracted, but we use classical mathematics imported from HOL, and a connection to some of HOL's definitional packages, for parts of the proof that do not contribute to the program.

1 Introduction

Interactive theorem provers typically require large libraries of formalized mathematics in order to be effective verification tools. Building these libraries is a time consuming and tedious activity, and is in large part duplicated effort, since, for example, many verification problems require similar theories of basic data types (integers, lists, bit-vectors, ...) no matter what system they are being formalized in.

An obvious approach to avoiding this duplication of effort is to share mathematics between different systems. However, it is not obvious that this is practical. Existing systems such as Nuprl [3], HOL [7], PVS [13], Isabelle [14], Coq [4] and the Boyer/Moore system [1], have different logics and different ways of automating reasoning. Mathematics is represented differently, and the exact syntactic form of the mathematics, as well as the particular choices of definitions and lemmas, is often influenced by the kind of automated reasoning being used.

In this paper we show that this approach is practical by developing a significant piece of formal mathematics in a hybrid of two different interactive systems. Using a connection between HOL90 [8] and Nuprl, we have constructed proofs which draw on mathematics developed in both systems. The proofs were done in Nuprl, using imported libraries of HOL mathematics, and using a connection between Nuprl and some of HOL's packages for adding constants, axioms and theorems that implement definitions of inductively defined predicates and ML-style recursive datatypes.

With this example we aim to show that despite the numerous significant differences between the logics of Nuprl and HOL, it is possible to effectively

[⋆] In *Proceedings of the 14th International Conference on Automated Deduction, July 1997.*

combine the two systems. One of the most serious differences is that Nuprl's logic is constructive, while HOL's is a classical set theory. In our example, we constructively prove a theorem in Nuprl, from which a program can be extracted, but we use classical mathematics from HOL for parts of the proof that do not contribute to the program.

The basic connection between HOL and Nuprl is at the level of theories. An HOL theory consists of type constants, individual constants, axioms and theorems. An HOL theory is imported into Nuprl via *interpretation*, whereby particular Nuprl objects are associated with the theory constants, the axioms are proven true of these objects, and then the theorems from the theory are declared to be Nuprl theorems.

In [10] we gave a new semantics for Nuprl that justifies an extension in which HOL's classical type theory (and other classical set/type theories) can be directly embedded. The extended logic is classical, but proofs that use only constructive reasoning still yield executable programs. In [9] we described the basic mechanism for importing an HOL theory into Nuprl and imported a few theories "by hand" as illustration.

The current work extends [9] as follows.

- We have added automated support for interpreting theories and updating Nuprl's automated reasoners to use the imported theories.
- The core of HOL90's standard library (i.e. the theory "HOL" and all of its ancestors) was imported, as well as an extensive theory of lists ("List"). This involved giving constructive implementations of the HOL theories that support defining recursive data types.
- We added a facility for using, in Nuprl, HOL's packages for defining recursive types and for making inductive definitions. In particular, a theory fragment generated in HOL by one of the packages can be imported and interpreted with very little intervention required from the user.
- In [9], the theorems in an imported theory were not directly usable in Nuprl proofs. However, we argued in [9] that it should almost always be possible to rewrite the theorems into a more usable form. We have implemented a procedure that automatically does this rewriting.
- Most importantly, we show that all of this is practical by applying it to a substantial example.

The original motivation for this line of work was to make Nuprl a more practical tool for software verification. Nuprl has a number of advantageous features, including a highly expressive type theory, the ability to extract programs from proofs, and a sophisticated user interface, but one thing that it lacks is the kind of large libraries of formal mathematics that are required for industrial-strength verification. The HOL user community has been steadily accumulating such libraries for at least 10 years, and it is now possible for Nuprl to draw on them. We are currently using Nuprl/HOL to formalize a correctness proof of a complex cache-coherency protocol [5].

Our example is a constructive proof of normalizability for a simply typed combinatory calculus. The proof yields a program that computes normal forms.

In addition, we prove that normal forms are unique. The formal mathematics in this example can be classified into three categories.

Imported from HOL. In addition to the standard library, we have imported an HOL theory of combinatory logic and a theory of minimal intuitionistic logic [2]. These theories define recursive types of the terms and types of combinatory logic, and give a number of inductively defined predicates, such as the reduction relation for terms. The imported theorems include support for reasoning about these types and predicates, and also the Church-Rosser theorem for reduction. In addition, we used HOL to define two new inductive predicates for head reduction, and imported these.

Classical mathematics in Nuprl. The bulk of the normalization argument was conducted using classical principles, building on the imported HOL mathematics. We first prove *classically* that every typed term has a normal form. The proof is based on a casting of Tait's computability argument in terms of semantics of a type system. This idea has been used to give an elegant proof of strong normalization of system F [11]. Nuprl can extract a "program" from our classical proof, but it will contain a term that represents an uncomputable "oracle" associated with the law of the excluded middle, and so the extracted object will not be executable. The uniqueness of normal forms is also proved classically.

Constructive mathematics in Nuprl. In interpreting the HOL theories, we gave constructive implementations of all the types they contained. In particular, HOL recursive types were given computationally reasonable implementations, and all the operations over them were given computable implementations. Doing this for all the imported theories drew heavily on the constructive theories in Nuprl's own standard library. Finally, we proved a theorem that constructivizes the normalization result, and from this proof we can extract an executable program.

As far as we know, there have been no other examples of theorem-proving where mathematics from two different interactive theorem provers has been combined. There have been numerous links between interactive theorem provers and other systems, but these have all been procedural: when a subgoal has the right form, it can be shipped off to a decision procedure or another system. Typical of links of this kind are the various connections between interactive provers and model-checking programs, for example [13] to cite one of many. Also, see [15] for a link between HOL and a resolution theorem prover.

2 The Connection between Nuprl and HOL

In this section we give an overview of the connection between HOL and Nuprl. Before proceeding to the overview, we give a very brief description of Nuprl. Formal mathematics in Nuprl is organized in a single library, which is broken into files simulating a theory structure. Library objects can be definitions, display forms, theorems, comments or objects containing ML code. Definitions define new operators, possibly with binding structure, in terms of existing Nuprl terms and previously defined operators. Display forms provide notations for defined and primitive operators. These notations need not be parsable since Nuprl uses structure editors. Theorems have tree structured proofs, possibly incomplete. Each node has a sequent, and represents an inference step. The step is justified either by a primitive rule, or by a *tactic*. Nuprl's notion of tactic is derived from that of LCF [6], as is HOL's.

2.1 Importing a Theory

The basis of the connection between Nuprl and HOL is theory importation. Mathematics in HOL is organized into a hierarchy of theories. We show how an HOL theory can be imported into Nuprl by giving a simple example.

The following HOL theory introduces a new type *one* with a single element also named *one*. HOL uses the symbols `?` and `!` for existential and universal quantification; `?!` is "exists-unique", and `\` is λ-abstraction. `@` is the "select", or "choose", operator which chooses an arbitrary value satisfying a predicate (here the universally true predicate).

```
Parents:        bool
Type constants: one 0
Term constants: one: one
Axioms:
Definitions:    one_TY_DEF |- ?rep. TYPE_DEFINITION (\b. b) rep
                one_DEF |- one = (@x. T)
Theorems:       one_axiom |- !f g. f = g
                one |- !v. v = one
                one_Axiom |- !e. ?!fn. fn one = e
```

This presentation omits the types of variables. For example, the type of `rep` in `one_TY_DEF` is `one -> bool`. This theory can be thought of semantically: ignoring the parent theory `bool`, if we associate a set with the type constant, and a member of the set to the term constant, and if all the formulas in the "axioms" and "definitions" sections of the theory hold for these objects, then all the theorems are true. In general, the semantics of a theory is parameterized by interpretations of all ancestor theories.

The old semantics of Nuprl was a term model based on an extended untyped λ-calculus, and the programs of this language are the basic objects one reasons about in Nuprl. Nevertheless, we have been able to construct a set-theoretic semantics [10] in which the meaning of each type is an ordinary set, and any program which is a member of a type can be given a meaning as a member of the set. In particular, the meaning of a function type $A \to B$ in Nuprl is the set of all set-theoretic functions from A to B. The collection of all sets that can be meanings for types can also be used to give the usual semantics for HOL.

As a result, we can directly apply the semantics of HOL theories to Nuprl. So, for the example theory above, if we associate any *Nuprl type* with the type constant *one*, and associate any member of that Nuprl type with the term constant *one*, and if the axioms/definitions are true for these objects, then the theorems are also true. Thus to import a theory, one *interprets* the type constants with Nuprl types and the term constants with members of the appropriate types, and then proves the formulas in the axioms/definitions sections. When this is done, the theorems can then all be accepted immediately as Nuprl theorems. Type-checking is undecidable in Nuprl, so the well-typedness of terms must be proven explicitly. This means that in addition to the axioms/definitions, it is also required to prove that each object associated with a type constant is a (non-empty) type, and that each object associated with a term constant has the type specified in HOL.

We now describe the implementation of this idea for theory importation by describing the steps one takes to import a theory.

The first step is to run HOL, load the desired theory into it, and then execute a function that writes out a file containing a Nuprl-readable version, called the *reference copy*, of the HOL theory, taking care of name conflicts that might arise because of Nuprl's single namespace.

The next step is to invoke a function within Nuprl that reads the reference copy of the HOL theory into Nuprl and then installs the theory into Nuprl's library structure. For each theory constant, a definition with an empty right hand side is created, and also a "well-formedness" theorem stating that the defined object has the appropriate type. The HOL definitions, axioms and theorems all are translated to Nuprl theorems. All translations of theorems are marked as unproven. Let us call the well-formedness theorems, together with the imported copies of the HOL axioms and definitions, the *proof obligations* of the theory.

The next step is to interpret the theory. This means supplying right hand sides for all the definitions, and then proving all the proof obligations. There is considerable automated support for constructing interpretations. For example, if an imported HOL constant is to be defined directly in terms of an existing Nuprl operator, then a single function can be used to complete the definition, prove the well-formedness theorem (always automatic in all our examples), and create library annotations that have the effect of adding the definitional rewrite to Nuprl's simplifier tactic. There is a variant of this function for the case when there is no existing operator, but the user wishes to create one, and for the case where the user wishes to define the HOL constant to be a particular Nuprl term but does not wish to make an additional Nuprl definition for the term. Whether or not to make an additional definition depends on whether the object will be used in later Nuprl work. If so, it is usually important to make a definition that adheres to Nuprl conventions, which is something the direct definitions of imported constants cannot do.

The final step is to call a function which "proves," and then rewrites, the imported theorems. The "proofs" are done by a tactic that refers to the reference copy of the theory in which the theorem to be proved resides, and also the reference copies of all ancestor theories. If for each such theory, all the definitions are complete, and the proofs of all the proof obligations have been completed, then the tactic marks the theorem as proven (using an unsound system feature). We do not translate the HOL proofs to Nuprl proofs; the soundness of marking the theorems as proven follows from the semantic argument sketched earlier.

In the rewriting stage of the final step, equational rewriting is used to derive from each imported theorem a new theorem that is more appropriate for application in subsequent Nuprl work. The rewriting is all formally justified within Nuprl, and applied through the tactic mechanism. We will not enumerate all the kinds of rewrites done, but will just give an example from list theory. The following is an imported theorem stating that a non-empty list is a cons. The names of all imported constants have an "h" prepended to avoid conflicts with Nuprl objects. The outermost quantifier quantifiers over the type S of all (small) non-empty types (this quantifier is implicit in HOL).

```
∀'a:S ↑(hall (λl:hlist('a).
                himplies (hnot (hnull l))
                         (hequal (hcons (hhd l) (htl l)) l)))
```

Note that, apart from the outermost quantifier, the logical connectives themselves are imported constants (introduced in ancestor theories of the theory containing the theorem). The new theorem generated from this is

```
∀'a:S. ∀l:'a List.  ¬mt(l) ⇒ hd(l)::tl(l) = l.
```

The logical connectives in HOL are all boolean-valued functions, possibly taking functional arguments, as in the case of the quantifiers. The interpretations of these connectives use boolean logic defined within Nuprl. The boolean connectives are rewritten in the second theorem to Nuprl's normal logical connectives, which are defined using a propositions-as-types correspondence. The operator ↑ in the imported theorem coerces a boolean into a Nuprl proposition. The imported list type is defined to be Nuprl's list type, and the imported tail function is defined to be Nuprl's tail function. Note however that htl is *applied*, as a function, to its argument, while the Nuprl tl is a defined operator with a single operand (Nuprl also has an operator for function application, of course). We have used a notational device to suppress type arguments in the (pre-rewrite) imported theorem. Each of the imported constants in the theorem actually has at least one type argument. In the rewritten theory, there are no hidden type arguments (the Nuprl operations are "implicitly polymorphic").

The most interesting point in this translation is the function for head of a list. In HOL, this is a *total* function on lists. When we import it into Nuprl, we must prove that the interpretation returns a value on every list, empty or not. Since hhd is polymorphic, given an arbitrary type and the empty list as an argument, it must choose some arbitrary member of the type as output. Thus we must give hhd a nonconstructive definition in Nuprl. However, we can prove that this function is the same as Nuprl's hd when the list is non-empty. This gives us a conditional rewrite which goes through for this example theorem.

In general, the rewriting of imported theorems is completely automatic, except for cases like hhd, where we may occasionally be left with a subgoal to verify the condition of some conditional rewrite. Usually such conditions are proven automatically.

The proofs of the new rewritten theorems are highly non-constructive, and the "program" extracted will often contain uncomputable operators. In [9] we outlined a simple scheme whereby the user could mark theorems as constructive, and the system would guarantee that the extracted program would not contain any uncomputable operators. Since most of the steps in constructive proofs do not contribute to the extracted program, imported HOL facts with non-constructive proofs could still be used. We have not yet implemented this scheme, and it is currently left up to the user to ensure that classical mathematics is not used inappropriately in constructive proofs.

2.2 Importing the Standard Theories

As part of the core of HOL's standard library, we have imported theories of booleans, pairing, disjoint union, arithmetic, lists, trees and labeled trees. Almost all the work done by the user in importing a theory is in proving the proof obligations. The vast majority of theories have no "axioms" section, and well-formedness theorems are almost always done automatically. There are two kinds of proof obligations arising from the "definitions" section of an HOL theory. The first kind are the axioms about the term constants. These axioms all look like recursive equational definitions and are almost always trivial to prove using rewriting.

The only possibly non-trivial work is in proving the definitional axioms about the type constants. In order to guarantee that every theory is consistent, i.e. has a model, the only axioms one is allowed to assume in HOL about a new type constant is that it is isomorphic to some non-empty subset of an existing type. For example, in the theory *one* above, there is the definitional axiom `one_TY_DEF` which says that there is a one-to-one function `rep`, from `one` to the type `bool` of booleans, whose range is the subset of the booleans satisfying `\b. b`, *i.e.* the singleton consisting of `T`.

One is also allowed in HOL to introduce term constants standing for a pair of functions giving the isomorphism. This mechanism essentially forces the HOL user (usually via automated packages) to specify an implementation in HOL of each new type introduced. The isomorphism pair can be used to define operations over the type.

Proving the definitional axioms for types involves showing that the definition given in Nuprl for the type is isomorphic to the HOL implementation. Since HOL uses set-theoretic style encodings (using, *e.g.*, equivalence classes) for basic type constructors such as product and disjoint union, and since there was no interest in giving computational appropriate implementations, the Nuprl implementations are often quite different and the proofs of equivalence can be quite non-trivial.

However, the hardest work in proving equivalence is in the early HOL theories. In later theories, the HOL implementations of data types are typically in terms of data types defined earlier, and are more likely to be directly usable as Nuprl implementations. In particular, equivalence is proved automatically in Nuprl for any new type introduced using HOL's recursive type package (see below).

The two most difficult theories to prove equivalence for were *tree* and *ltree*, which form the basis for HOL's recursive type package. These theories introduce the types of finitely-branching trees and finitely-branching labeled trees, respectively. In Nuprl, we can give a natural definition of labeled trees as a fixed point

$$\mathit{ltree}(A) = A \times (\mathit{ltree}(A)\ \mathit{list})$$

where A is the type of labels. Nuprl does not have a recursive type constructor, but the new semantics justifies a form of indexed union of types, and the fixed-point can be constructed as $\cup_{n\in\omega}F^{(n)}(\emptyset)$, where $\emptyset$ is Nuprl's empty type, $F^{(n)}$ is the n-fold composition of F with itself, and F is the type functional mapping T to $A \times (T\ \mathit{list})$.

To implement the HOL type of unlabeled trees, we use *ltree*(*unit*) where *unit* is a one-element type. We prove that this is isomorphic to the HOL implementation, which encodes trees as natural numbers using exponentiation and other arithmetic operators. The HOL implementation of a labeled tree is a pair consisting of an unlabeled tree and a list of labels whose length is the number of nodes in the tree.

2.3 Support for HOL Definitional Packages

HOL has a package for making a limited form of ML-style recursive type definitions. The package takes as input a specification of the type that has a syntactic form close to ML's. The result of running the package in HOL is an extension to the current theory, including theorems and tactics for reasoning about the type. There is also a package for making inductive definitions of predicates and relations. We describe how we have automated the importation of the results of these packages via examples.

Recursive Types An ML-style definition of the type of terms of combinatory logic is

$$cl \ = \ s \ | \ k \ | \ \#\ \mathit{of}\ cl \times cl$$

where $\#$ is the constructor for application terms. The HOL recursive type package implements this as a "subset" of the type of labeled trees, for a particular choice of label type. The label type is built from the unit type *one* and disjoint union, and has as many elements as there are constructors in the type definition. For this example, the package introduces one new type constant, `cl`, and five new term constants:

```
REP_cl: cl -> (one + one + one) ltree
ABS_cl: (one + one + one) ltree -> cl
s : cl      k : cl      # : cl -> cl -> cl
```

where the first two are used to axiomatize the isomorphism between `cl` and the elements of the representation type that satisfy a predicate specifying a correspondence between the label of a node and the number of children it has. The package also introduces five axioms. The first two specify the isomorphism. The other three define the constructors. For example, we have

```
|- s = ABS_cl (Node (INL one) [])
|- !c1 c2. c1 # c2 = ABS_cl (Node (INR (INR one)) [REP_cl c1; REP_cl c2])
```

where `Node` builds a tree node from a label and a list of trees.

When the theory containing these objects is imported into Nuprl, we execute the function *rec_type*, giving it as arguments the name of the imported recursive type and the names of the constructors. It then does the following. It first determines the label type and defines a Nuprl type analogous to the HOL implementation, using the Nuprl *ltree* discussed above and Nuprl's subtype constructor. All the components for this definition are found in the imported objects. Next, it completes the definitions of the `ABS` and `REP` functions. The latter is the identity function, but the former is the identity only on a subset of its domain, and is extended to a total function using an uncomputable test. It then gives constructive definitions of the constructors. These are obtained essentially by taking the corresponding HOL definitional axioms and erasing the `ABS` and `REP` functions. Finally, it proves all the proof obligations.

This is all completely automatic, with one exception. For technical reasons, a recursive type must be shown non-empty. These proofs are easy, but we have not automated them yet. Note that HOL does all the syntactic work of translating the ML-style specification into the appropriate label type, constructor definitions and set of tailored theorems for induction, case analysis etc.

An HOL theory with a recursive type definition contains a number of useful theorems for reasoning about the type. The imported versions of these theorems can be used for reasoning about the Nuprl implementations of the type. We have also written a number of tactics for effectively applying these theorems. Examples are given in the next section.

We have also proven a constructive induction principle for recursive types, although we did not use it in our example. Because Nuprl has subtypes, we can directly express the induction principle for all recursive types (subtypes of *ltree(A)* for some A) in a single lemma. An HOL approximation to this would need to refer to isomorphism pairs.

Inductively Defined Predicates The HOL package for inductive predicate definitions defined in [12] automates the derivation of certain inductive definitions, and generates and proves theorems needed to reason about them. The package takes a set of rules as input, generates an HOL term defining the inductive predicate, and proves a theorem stating that this predicate is the least such relation closed under the rules. An induction theorem as well as theorems stating that the rules hold (*i.e.*, that the premises imply the conclusion) follow directly from this theorem. A strong induction theorem and an exhaustive case analysis theorem are also generated and proved. A good part of the work of the package is the syntactic translation from rules to definition and theorems. A typical example taken from [12] and also used later in our proofs is the following set of rules denoting the reflexive transitive closure of a relation R.

$$\frac{R(x,y)}{R^*(x,y)} \qquad \frac{}{R^*(x,x)} \qquad \frac{R^*(x,z) \qquad R^*(z,y)}{R^*(x,y)}$$

The package generates the following definitional axiom.

$$\begin{array}{l}(\forall R\,x\,y.\ R\ x\ y \supset \mathsf{Rtc}\ R\ x\ y)\wedge\\ (\forall R\,x.\ \mathsf{Rtc}\ R\ x\ x)\wedge\\ (\forall R\,x\,y.\ (\exists z.\ \mathsf{Rtc}\ R\ x\ z \wedge \mathsf{Rtc}\ R\ z\ y) \supset \mathsf{Rtc}\ R\ x\ y)\wedge\\ (\forall R\,P.\ (\forall x\,y.\ R\ x\ y \supset P\ x\ y)\ \wedge (\forall x.\ P\ x\ x)\ \wedge\\ \qquad (\forall x\,y.\ (\exists z.\ P\ x\ z \wedge P\ z\ y) \supset P\ x\ y)) \supset \forall x\,y.\ \mathsf{Rtc}\ R\ x\ y \supset P\ x\ y)\end{array}$$

We have written a procedure *new_ind_def* that is used after a theory containing inductively defined predicates is imported into Nuprl. This procedure first derives from the imported version of the above axiom a Nuprl definition of the inductive predicate, and then completes well-formedness proofs and automates the proof of the axiom. This tactic usually completes the proof; when it doesn't, the proof is easily completed by user-guided instantiation of existential quantifiers. Examples are given in the next section of theorems generated for reasoning about these types as well as tactics we have written to automate their application.

3 Normalization in the Simply-Typed SK Calculus

Our proof has two main components. First, we define the *values* of the SK combinator calculus, show that reduction of a term can result in at most one value, and then define a notion of complete head reduction and show it results in values. Second, we show that all typed terms of the SK calculus are normalizable with respect to head reduction. The first component uses classical reasoning. The second is constructive, and results in a normalization program, though classical reasoning is used for parts of the proof that do not contribute to the program. Before describing the normalization proof, we describe some of the mathematics that we import from HOL.

The HOL theories of combinatory logic "CL" and minimal intuitionistic logic "MIL" that we import from HOL use the recursive types package to define the syntax of both the terms of combinatory logic (illustrated in the previous section) and of simple types. The syntax of types, in ML notation, is

$$ty(\alpha) = \ g\ of\ \alpha\ \mid\ mil\_fun\ of\ ty(\alpha) \times\ ty(\alpha)$$

where α is a type variable that can be thought of as a set of base or ground types. We introduce special notation, using Nuprl's display features, for some of the Nuprl versions of the constructors of the types *cl* and *ty(α)*. In particular, *s* becomes *S* in Nuprl (not to be confused with the type of all non-empty types introduced in the previous section), *k* becomes *K*, *e1 # e2* becomes simply *e1 e2*, and *mil_fun (t1,t2)* becomes *t1 → t2*.

Theorems generated by the recursive types package include existence and uniqueness of a primitive recursion operator, induction, case analysis, and injectivity and distinctness of the constants. As discussed in the previous section, when these theorems are imported they are rewritten to a more "Nuprl-friendly" form. These rewritten theorems are used extensively in our normalization proof. We list them for `cl` below except the one for primitive recursion. Instead, we give the primitive recursion theorem for `ty` since we use it below (though for conciseness, we elide the clause expressing uniqueness of the recursion operator.)

```
cl_induct  ∀P:cl → 𝔹. ↑(P S) ∧ ↑(P K) ∧
 (∀c1,c2:cl.  ↑(P c1) ∧ ↑(P c2) ⇒ ↑(P (c1 c2))) ⇒ (∀c:cl. ↑(P c))
cl_cases   ∀c:cl. c = S ∨ c = K ∨ (∃c1,c2:cl. c = c1 c2)
ap11  ∀c1,c2,c1',c2':cl. c1 c2 = c1' c2' ⟺ c1 = c1' ∧ c2 = c2'
cl_distinct  ¬(S = K) ∧ (∀c2,c1:cl. ¬(S = c1 c2)) ∧
 (∀c2,c1:cl. ¬(K = c1 c2)) ∧ ¬(K = S) ∧
 (∀c2,c1:cl. ¬(c1 c2 = S)) ∧ (∀c2,c1:cl. ¬(c1 c2 = K))
mil_fun_rec
 ∀'a1,'a:S. ∀f0:'a → 'a1. ∀f1:'a1 → 'a1 → ty('a) → ty('a) → 'a1.
  (∃fn:ty('a) → 'a1. (∀x:'a. fn g(x) = f0 x)
    ∧ (∀t1,t2:ty('a).  fn (t1 →  t2) = f1 (fn t1) (fn t2) t1 t2))
```

Note that in the first theorem, we quantify over boolean valued functions instead of Nuprl predicates, which have values in type $\mathbb{P}_1$. Applications of the function in the theorem are coerced to Nuprl propositions using ↑. We could have strengthened our rewriting to translate boolean-valued functions to proposition-valued functions, but the gain would be not be drastic since our tactics automatically handle these coercions when using lemmas like this one. Note that there is duplication in the `cl_distinct` theorem because equality is symmetric; this form of the theorem is convenient for tactics. Recall that the constant `S` is overloaded. In `mil_fun_rec`, it denotes HOL types (all non-empty types), while all other occurrences in the above theorems denote the *s* combinator.

The HOL inductive predicate package was used to define the contraction relation (denoted ⟶) in the HOL theory CL. This relation has four rules as given in the first four imported theorems below. We also give the imported theorems for strong induction and case analysis for illustration.

```
c_k     ∀x,y:cl. K x y ⟶ x
c_s     ∀x,y,z:cl. S x y z ⟶ x z (y z)
c_app1  ∀x,y:cl. x ⟶ y ⇒ (∀z:cl. x z ⟶ y z)
c_app2  ∀x,y:cl. x ⟶ y ⇒ (∀z:cl. z x ⟶ z y)
csind   ∀P:cl → cl → 𝔹. (∀x,y:cl. ↑(P (K x y) x))
  ∧ (∀x,y,z:cl. ↑(P (S x y z) (x z (y z))))
  ∧ (∀x,y:cl. x ⟶ y ∧ ↑(P x y) ⇒ (∀z:cl. ↑(P (x z) (y z))))
  ∧ (∀x,y:cl. x ⟶ y ∧ ↑(P x y) ⇒ (∀z:cl. ↑(P (z x) (z y))))
  ⇒ (∀U,V:cl. U ⟶ V ⇒ ↑(P U V))
ccases  ∀U,V:cl. U ⟶ V ⟺ (∃y:cl. U = K V y)
  ∨ (∃x,y,z:cl. U = S x y z ∧ V = x z (y z))
  ∨ (∃x,y,z:cl. U = x z ∧ V = y z ∧ x ⟶ y)
  ∨ (∃x,y,z:cl. U = z x ∧ V = z y ∧ x ⟶ y)
```

The reduction relation (denoted *⟶) is defined as the reflexive transitive closure of this relation, using the definition of Rtc in the previous section.

The CL theory also defines the Church Rosser property, and proves that the reduction relation has this property. Many auxiliary definitions and lemmas are needed in the HOL proof, including a notion of parallel reduction.

Type assignment is defined as an inductive predicate in the MIL theory. In particular, `e : t` means that combinatory logic term `e` has simple type `t`. We omit the details.

We now describe some of the classical mathematics that we build in Nuprl on top of the imported HOL math. We define a head reduction relation (denoted ↓↓) using the HOL inductive predicate definition package. The following nine imported theorems illustrate its definition.

```
K ↓↓ K          S ↓↓ S
∀x,x':cl. x ↓↓ x' ⇒ K x ↓↓ K x'
∀x,x':cl. x ↓↓ x' ⇒ S x ↓↓ S x'
∀x,x':cl. x ↓↓ x' ⇒ (∀y:cl. K x y ↓↓ x')
∀x,x',y,y':cl. x ↓↓ x' ∧ y ↓↓ y' ⇒ S x y ↓↓ S x' y'
∀x,z,v:cl. (∃x':cl. x ↓↓ x' ∧ x' z ↓↓ v) ⇒ (∀y:cl. K x y z ↓↓ v)
∀x,y,z,v:cl. (∃x',y':cl. x  ↓↓ x' ∧ y ↓↓  y' ∧ x' z (y' z) ↓↓ v)
               ⇒ S x y z ↓↓ v
∀x1,x2,x3,x4,x5,v:cl. (∃y:cl. x1 x2 x3 x4 ↓↓ y ∧ y x5 ↓↓ v)
                        ⇒ x1 x2 x3 x4 x5 ↓↓ v
```

Note that this relation is defined using a natural semantics style in which if (e ↓↓ v) holds then v is a value. (We prove this fact below.) Also note that these rules are deterministic and syntax directed: if (e ↓↓ v) holds, then it can only be derived by one rule, determined by the form of e. The normalization program discussed below implements the procedure implicit in these rules.

We define head reduction using HOL in order to take advantage of the automation provided by the inductive definition package. Although we used HOL directly, then imported the resulting theory, and interpreted it using *new_ind_def*, it would not be difficult to automate the entire process.

We also define a head reduction relation that takes an additional natural number argument which records the size of the proof, *i.e.*, (e ↓↓{n} v) if and only if (e ↓↓ v) follows by a derivation of n steps. This relation is imported similarly to ↓↓ and is needed to constructivize our normalization result.

Before describing the Nuprl normalization proof, we first discuss the tactics that we have implemented to help automate these proofs using theorems generated by either the recursive types or the inductive definition packages of HOL. For instance, we have written general tactics for applying theorems for induction (*e.g.* csind) and case analysis (*e.g.* ccases). The induction tactic, given a term inhabiting a recursively defined type, or an instance of an inductively defined predicate, finds and applies the appropriate induction theorem. We have also written a tactic to help with reasoning about equality of terms inhabiting recursively defined types. Given a goal with a hypothesis of the form e = e' ∈ T, if the outermost constructors of the two terms are different, the tactic uses the imported distinctness theorem (*e.g.* cl_distinct) to complete a proof by contradiction. If they are the same, it uses the imported injectivity lemmas (*e.g.* ap11) to conclude equality of subterms, and then proceeds recursively.

This tactic was used, for example, to prove inversion lemmas. Given that an inductively defined relation holds of some particular terms, inversion lemmas can be used to conclude which rule was last applied based on the form of the terms and conclude that the relation holds for the rule premises. For example, an inversion lemma for the h_k1 rule is:

```
h_k1_inv  ∀u,v:cl.  K u ↓↓ v ⇒ (∃v':cl. v = K v' ∧ u ↓↓ v')
```

Such lemmas are proved by case analysis, and most cases can be completed automatically, the remaining cases required only slight user assistance.

In Nuprl, we define a *normal form* as a term on the right of the ↓↓ relation, that is, `norm(e) == ∃v:cl. e ↓↓ v`. We define a *value* to be a term which does not contract and then show that normal forms are values. The proof is by induction on the ↓↓ relation, using the imported strong induction principle, where each subcase is then proved by case analysis on the ⟶ relation. We further show that if `e ↓↓ v`, then `e` reduces to `v`, and (using the imported Church-Rosser theorem) any value it reduces to is equal to `v`.

We now describe the proof of the following theorem that states that every well-typed term is normalizable.

```
type_norm   ∀'a:S. ∀e:cl. ∀t:ty('a).  e : t ⇒ norm(e)
```

The proof adapts Tait's notion of computability to combinatory logic. The proof is classical and builds on imported definitions and theorems. Computability is defined using primitive recursion over simple types. A term `e` is computable at type `t`, written `(e ∈ t)`, if `e` is normalizable, and if whenever `t` is an arrow type `t1 → t2` and `e'` is a computable term at type `t1`, then `(e e')` is a computable term at type `t2`. The Nuprl definition of computability uses `mil_rec_fun` given above. Normalizability is then proved via the following two lemmas.

```
type_comp  ∀'a:S. ∀e:cl. ∀t:ty('a).  e : t ⇒ e ∈ t
comp_norm  ∀'a:S. ∀t:ty('a). ∀e:cl.  e ∈ t ⇒ norm(e)
```

The latter follows from the definition of computability, and uses case analysis on types. The former is proved by induction on types, where the induction step uses case analysis on terms. The application case follows directly from the induction hypothesis, while the `S` and `K` cases use a variety of lemmas stating properties of head reduction and computability. We omit these lemmas and simply note that their proofs require various inductions and case analyses on both recursive types and inductive predicates, and also draw heavily on inversion lemmas, as well as imported distinctness and injectivity lemmas.

We can now prove the main constructive result:

```
norm_thm  ∀'a:S. ∀e:{x:cl| ∃T:ty('a). x : T} .  ∃v:cl. ↓(e ↓↓ v)
```

We have used Nuprl's subtype constructor twice in the statement of this theorem to guarantee that the extracted program does not manipulate witnessing data for the typing relation. The type of `e` above is the set of all terms in `cl` that have a type. The members of this subtype are just terms, not, e.g. pairs of terms together with evidence that they have a type. Similarly, in the conclusion of the lemma, we have used the "squash operator" ↓, which is defined using subtypes and produces a type that has a single fixed member if its argument is true and is empty otherwise. Thus any extracted program will essentially map terms to terms.

Given `type_norm`, we can reduce the proof of the main theorem (`norm_thm`) to the following goal.

```
hnf_exists  ∀e:{x:cl| ∃x':cl. x ↓↓ x'} . ∃v:cl. ↓(e ↓↓ v)
```

In proving this theorem, we must take care not to use any facts imported from HOL, or any facts proven using the non-constructive extensions of Nuprl's logic, in the part of the proof that contributes to the program. Our proof is straightforward. We explicitly write a program that computes the head normal form and use this program to provide a witness for the existential quantifier. The program is written in a natural ML-style, using general recursion and pattern-matching. All the work is showing termination of the program. We do this by using the characterization

```
hnred_iff_hred  ∀e,v:cl.  (∃n:ℕ. e ↓↓{n} v) ⟺ e ↓↓ v
```

then proving by induction on `n` that the program terminates on argument `e` whenever there exists a `v` such that `e ↓↓ v`.

4 Conclusion

HOL and Nuprl have a similar approach to automated reasoning. Most of the work in proving a theorem is applying collections of facts using general machinery, such as resolution and rewriting tactics, that both systems possess. Thus the vast majority of imported facts can be effectively applied using the normal Nuprl tactics. However, some facts, such as the "distinctness" theorems associated with recursive types, are applied in HOL using special purpose tactics. We had to duplicate some of these tactics in Nuprl in order to effectively apply these facts.

The use of program extraction was somewhat minimal in our example. Most of the argument was classical, and at the end we explicitly wrote the program that gave the constructive content of the main normalization result. A more interesting use of extraction would have been possible if we had reproved some of the imported facts using constructive induction principles. This situation, where imported facts have some interesting computational content, is atypical for software/hardware verification, where most of the work goes into verifying properties, such as equalities and inequalities, that have no non-trivial constructive content.

Our use of a version of head-reduction incorporating derivation size was for technical reasons. One deficiency with Nuprl's type theory is that termination of recursive programs can only be proved via built-in induction principles. In particular, we could not do induction over the imported head-reduction predicate.

Because of Nuprl's richer logic, it would difficult to import mathematics from Nuprl into HOL. Of course, we could restrict importation to a HOL-like subset of Nuprl. Another possibility is a "HOL-mode" for Nuprl, where HOL tactics can be called on goals in the HOL-like subset. Practically, Nuprl might have more to gain from making this kind of connection with PVS, which would bring more new procedural machinery to Nuprl, and which would apply to a much larger subset of Nuprl.

Since we developed supporting machinery at the same time as the normalization proof, it is difficult to say how long the latter took. Doing another similar proof using our infrastructure would probably take about a day.

References

1. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.
2. J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, August 1992.
3. R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
4. C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. The Coq Proof Assistant reference manual. Technical report, INRIA, 1995.
5. A. Felty and F. Stomp. A correctness proof of a cache coherence protocol. In *Proceedings of the 11th Annual Conference on Computer Assurance*, June 1996.
6. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.
7. M. J. C. Gordon and T. F. Melham. *Introduction to HOL—A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
8. HOL90. The HOL90 distribution. (available from ftp://ftp.research.bell-labs.com/dist/hol90).
9. D. J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 267–282, Berlin, 1996. Springer-Verlag.
10. D. J. Howe. Semantics foundations for embedding HOL in Nuprl. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101, Berlin, 1996. Springer-Verlag.
11. D. McAllester, J. Kučan, and D. Otth. A proof of strong normalization for F_2, F_ω and beyond. *Information and Computation*, 121(2):193–200, September 1995.
12. T. Melham. A package for inductive relation definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 350–357. IEEE Computer Society Press, 1992.
13. 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 Notes in Computer Science. Springer Verlag, 1996.
14. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.
15. K. Schneider, R. Kuma, and T. Kropf. Integrating a first-order automatic prover in the HOL environment. In *Proceedings of the 1991 International Tutorial and Workshop on the HOL Theorem Proving System and its Applications.* IEEE Computer Society Press, 1992.