Nuprl–Light: An Implementation Framework for Higher-Order Logics

Jason J. Hickey*

Cornell University, Ithaca, NY 14853, USA.
jyh@cs.cornell.edu.

1 Introduction

Recent developments in higher-order logics and theorem prover design have led to an explosion in the amount of mathematics and programming that has been formalized, and the theorem proving community is a faced with a new challenge—sharing and categorizing formalized mathematics from diverse systems. This mathematics is valuable—in many case many man-months, or even man-years, have been devoted to the development of these mathematical libraries. There is potential for more rapid advance if theorem provers of the future provide a means to relate logics formally, while providing adequate protection between logics with differing assumptions.

In this paper we describe *Nuprl-Light*, a descendent of the Nuprl [2] theorem prover, that addresses the issues of diversity and sharing by providing a modular, object-oriented framework for specifying, relating, and developing type theories and mathematical domains. The framework itself assumes (and provides) no type theory or logic, as in LF [4], which is why we call it an *implementation* framework. Instead, *Nuprl-Light* provides a meta-framework where logical frameworks such as LF, Nuprl, set theory, and other theories can be defined and developed. Since proof automation is such a critical part of theorem proving in these logics, the implementation framework is tied closely to a programming language (in this case *Caml-Light*) and the formal module system is tied closely to the programming language modules.

Like the Isabelle [9] generic theorem prover, *Nuprl-Light* uses generalized Horn clauses for logical specification. Indeed, specifications in *Nuprl-Light* appear quite similar to those in Isabelle. However, where Isabelle uses higher order unification and resolution, *Nuprl-Light* retains a tactic–tree [3] of LCF [8] style reasoning based on tactics and tacticals, and *Nuprl-Light* also allows theories to contain specifications of rewrites, using the computational congruence of Howe [7]. Like LF, the *Nuprl-Light* meta-logic also relies on the judgments–as–types principle (an extension of propositions–as–type), where proofs are terms that inhabit the clauses.

* Support for this research was provided by the Office of Naval Research through grant N00014-92-J-1764, from the National Science Foundation through grant CCR-9244739, from DARPA through grant 93-11-271, and from AASERT through grant N00014-95-1-0985.

The main departure from Isabelle and LF is in the module system. In *Nuprl-Light* modules have formal first-class signatures and implementations, providing the ability not only to specify multiple logics, but to relate them (using functors). In addition, modules in *Nuprl-Light* are object–oriented, providing the ability to extend type theories and their reasoning strategies incrementally. Taken together, these abilities provide a view of “theories–as–objects,” encouraging incremental and modular specification of theories.

These are the results we address with this system:

– An implementation framework for specifying and relating type theories, and their rules, theorems, and proofs.
– A method for constructing formal types from module signatures, and formal object for module implementations, based on recent theoretical work with very–dependent function types [6].
– An architecture for incrementally implementing algorithms for automated reasoning.
– A mechanism for generic shared tactics and derived rules.

2 The Framework

The formal system uses a term language to define formal objects such as numbers $(0, 1, 2, \ldots)$, functions $(\lambda x.b)$, dependent function spaces $(\Pi x{:}\, A.B)$, as well as sequents $(\Gamma \vdash \Delta)$. The framework itself attaches no meaning to these terms—that is the duty of the logic, not the framework, and so, for instance, the operator $+$ does not “perform” addition until that meaning is attached to it.

`signature`	::=	`theory_sig` *name* = *sig_stmts* `end`
`sig_stmt`	::=	`axiom` *name* $p_1 \ \ldots \ p_n$: *judgement*
	|	`rewrite` *name* $t_1 \Longleftrightarrow t_2$
	|	`include` *name*
	|	`declare` *term*
	|	`dform` *term* = *dform*
	|	*signature*
	|	*ML declaration*
`implementation`	::=	`theory` *name* = *thy_stmts* `end`
`thy_stmt`	::=	`axiom` *name* $p_1 \ldots p_n$: *judgement*
	|	`prim` *name* $p_0 \ldots p_n : (v_1{:}\, t_1) \cdots (v_{m-1}{:}\, t_{m-1}) = t : t_m$
	|	`thm` *name* $p_0 \ldots p_n : t_1 \ldots t_{m-1} = \mathit{tactic} : t_m$
	|	`include` *name*
	|	`primrw` *name* $t_1 \Longleftrightarrow t_2$
	|	`rwthm` *name* $t_1 \Longleftrightarrow t_2 = \mathit{tactic}$
	|	*ML implementation*

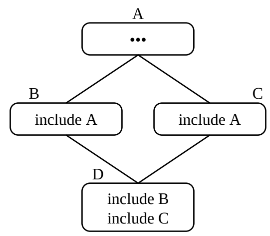
Mathematical theories are formulated as *theories*, which are an extension of the ML module system. Each theory has a *signature*, which specifies the rules and types of the theory, and an *implementation*, which contains proofs for the rules. The syntax for theories is shown in the Figure above.

A rule is declared in a theory signature with an `axiom` form. The declaration

$$\texttt{axiom}\ \mathit{name}\ p_1 \ldots p_n : t_1 \Rightarrow \cdots \Rightarrow t_m$$

specifies that the term t_m is true, if the antecedents $t_1, \ldots, t_{m-1}$ are true. The `prim` form is used to implement the rules that are primitive in the type theory, and for constructive logics, it also specifies the proof extract term. For instance, the and-introduction rule might be "implemented" as follows.

$$\texttt{prim}\ \mathit{and\_elim} : (a\colon \Gamma \vdash A)\ (b\colon \Gamma \vdash B) = (\langle a, b\rangle : \Gamma \vdash A \wedge B)$$



The theories are object-oriented, in the sense that a theory specifies a *class* that can inherit rules and implementations from other classes. All rules and theorems that are valid in a superclass remain valid in the subclass. Operationally, an `include` directive treats the included theory is if it were inlined in the module, with one exception: modules are inlined at most once (an implicit sharing constraint). In the diagram above, we describe a scenario where modules B and C both `include` module A, and module D `include`s both B and C. Only *one* copy of A is inlined, and the modules B and share the common implementation.

The `rewrite` form defines computational rewriting. For instance, the declaration,

$$\texttt{rewrite}\ \mathit{beta} : (\lambda x.M[x])\ N \Longleftrightarrow M[N],$$

defines beta equivalence. The `primrw` and `rwthm` correspond to the `prim` and `thm` forms, except that rewrites are assigned no proof extract, so the justification omits it.

A theory may also extend the formal language by declaring a new term. For instance, a module that defines number theory would extend the term language with a term that specifies addition using the `declare` form:

$$\texttt{declare add}\{\}(v_1; v_2).$$

Terms that are declared are associated with the module in which they are declared.

One of the key features of the framework is that theories and their signatures are first class. The framework provides a means to extract a formal type from a theory signature, and a formal object from its implementation. The general idea is to translate a module signature to a dependent record type, and translate its implementation to a record inhabiting that type. As usual, the framework does not assign meaning to a record and its type—that job is left to the type theory designer. However, in logics that are expressive enough to reason about dependent record types, it is expected that the normal record subtyping will be derivable. In the Nuprl type theory, record types are interpreted as very-dependent function types, where the functions range over the set of labels in the record, and the expected subtyping holds since a function with a larger domain can simulate a function with a smaller domain. A more complete description is given in Hickey [6].

3 Example

```
theory_sig ipl_implies_sig =
    include sequent_sig
    include ipl_false_sig
    declare A ⇒ B
    declare λx.b[x]
    declare M N (* application *)

    declare ¬A
    rewrite neg_rw : ¬A ⟺ (A ⇒ ⊥)

    axiom imp_intro a : (Γ, a: A ⊢ B) ⇒ (Γ ⊢ A ⇒ B)
    axiom imp_elim : (Γ, Δ ⊢ A) ⇒ (Γ, x: B, Δ ⊢ C) ⇒ (Γ, x: A ⇒ B, Δ ⊢ C)
end
```

For an example, we define a fragment intuitionistic propositional logic. In the Figure above, we show the fragment of the logic that defines implication. The fragment depends on the definition of *false* to define negation, as well as the definition of sequents, so it includes the theories `ipl_false_sig` and `sequent_sig`. This fragment *declares* the new term for implication, as well as a definition of negation, and then it specifies the rules for implication introduction and elimination.

Although we can provide a primitive implementation, it is also possible to derive an implementation by *relating* the logic to another—in other words, by giving a model in terms of another existing logic. In this example, shown in the Figure below, we show how to derive an implementation for `ipl_implies_sig` from the Nuprl type theory ITT. The Nuprl type theory contains IPL as a proper sub-theory, and the justification is quite straightforward. We interpret the IPL implication as the ITT implication (written `implies(A; B)` in the example), which allows the IPL rules to be justified from the ITT rules by unfolding the definition of the implication. Although this justification is technically trivial, the pattern for more complex justifications is similar–although the interpretations of the rules and symbols may be more complex.

```
theory ipl_implies =
    include ITT
    rewrite imp_def : (A ⇒ B) ⟺ implies(A, B)
    rewrite lam_def : λx.b[x] ⟺ lambda(x.b[x])
    rewrite apply_def : (M N) ⟺ apply(M; N)

    thm imp_intro a (Γ, a: A ⊢ B) =
        (unfold imp_def andthen itt_imp_intro) : (Γ ⊢ A ⇒ B)

    thm imp_elim (Γ, Δ ⊢ A) (Γ, x: B, Δ ⊢ C) =
        (unfold imp_def andthen itt_imp_elim) : (Γ, x: A ⇒ B, Δ ⊢ C)
end
```

4 Implementation

Nuprl-Light is implemented in *Caml-Light*, with a total of about 25,000 lines of source code to implement the theorem prover, and an additional 10,000 to specify the Nuprl type theory and implement the tactics. The heart of the implementation is a rewriting engine that is used both for computational rewrites, and proof refinements. Care was taken to make term rewriting efficient, and *Nuprl-Light* pre-compiles rewrite specifications to an intermediate language. This rewriting engine, together with abstract operations on terms, count for about 20% of the code. The rest of the code is devoted to algorithms for proof search, display printing, and file processing.

The Nuprl type theory is implemented as a collection of modules, one for each type constructor. One unexpected benefit of this coding is that with the use of derived rules the number of primitive inference rules needed to define the type theory and its type constructors has dropped by about a factor of five, since most of the standard type constructors can be derived from the very-dependent function type. Our plans for the future include further development of the tactic collection and improvements to proof search algorithms. The implementation and further information are available at the author's home page `http://www.cs.cornell.edu/home/jyh`.

References

1. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Conference on Logic in Computer Science*, pages 195–197, June 1987.
2. R.L. Constable et.al. *Implementing Mathematics in the NuPRL Proof Development System*. Prentice–Hall, 1986.
3. Timothy G. Griffin. *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems*. PhD thesis, Cornell University, 1988.
4. Rober Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1), January 1993.
5. Robert Harper and Mark Lillibridge. A type–theoretic approach to higher–order modules with sharing. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 123–137. ACM, January 1994.
6. Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the FOOL 3 home page at Williams College.
7. Douglas J. Howe. Equality in Lazy Computation Systems. In *Fourth Annual Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society Press, 1989.
8. Lawrence C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge Univ. Press, 1987.
9. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer LNCS 828, 1994.

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