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

Jason J. Hickey*

Department of Computer Science
Cornell University
Ithaca, NY 14853, USA.
Tel: (607) 255-1372
Email: jyh@cs.cornell.edu.

Abstract. In this paper we describe a new theorem prover architecture that is intended to facilitate mathematical sharing and modularity in formal mathematics and programming. This system provides an implementation framework in which multiple logics, including the Nuprl type theory and the Edinburgh Logical Framework (LF) can be specified, and even related. The system provides formal, object-oriented modules, in which multiple (perhaps mutually inconsistent) logics can be specified. Logical correctness is enforced and derived from module dependencies. Support is provided at a primitive level for modular proof automation.

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 [5] 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 [9], 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.

* 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.

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 [25] 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 [8] of LCF [24] style reasoning based on tactics and tacticals, and *Nuprl-Light* also allows theories to contain specifications of rewrites, using the computational congruence of Howe [15]. 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.

The main departure from Isabelle and LF is in the module system. In *Nuprl-Light* modules have 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 present in the paper:

- 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 [13].
- An architecture for incrementally implementing algorithms for automated reasoning.
- A mechanism for generic shared tactics and derived rules.

Nuprl-Light is implemented in *Caml-Light*, and we have implemented the Nuprl type theory and tactics, as well as a formalization of LF. The Nuprl formalization is modular, with a distinct module for each type constructor. The rules for automated reasoning, including type inference, and forward and backward chaining, have been specified separately for each type constructor, and the framework automatically constructs the type theory and its tactic collection.

In the next section we describe the framework more formally. Following that, we give an example and discuss the implementation.

2 The Framework

One of our goals in the the design of the framework has been to tie the formal system tightly with the underlying programming language used for its implementation to enhance the interplay between the formal and practical methodologies. The main supplement to the language is the definition of a formal term language, and a method for including and deriving formal assertions in the module calculus. In the following, we will assume a working knowledge of ML modules [10].

The presentation is in four parts. First we will cover the formal language and syntax, and then we we will cover the rules for combining and deriving formal judgments. Following the formal description, we cover the framework for proof automation, and give a few examples.

2.1 Formal module system

The formal system uses a term language that is distinct from the programming language. These terms include the usual objects such as numbers $(0, 1, 2, \ldots)$, functions $(\lambda x.b)$, dependent function spaces $(\Pi x\colon A.B)$, 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.

The underlying term language is uniform, based on the encoding of Allen et. al. [3], where every term has this form.

$$\texttt{opname}\{\mathit{params}\}(\mathit{bterms})$$

Every class of terms has a unique name (the *opname*). The parameters are used to specify the term constants, such as the numbers, and the *bterms* is a sequence of subterms with optional binding occurrences. The following table lists some examples of terms in this language. The print representation of these terms is defined by declaring *display forms*, which we do not discuss in this paper. In this paper we will use the print representation for terms, and the existence of a uniform encoding is understood.

$$\begin{array}{lr@{}l} \text{Term examples} & 0 \equiv{} & \texttt{number}\{0\}() \\ & 1 \equiv{} & \texttt{number}\{1\}() \\ & t_1 + t_2 \equiv{} & \texttt{add}\{\}(t_1; t_2) \\ & \lambda x.b \equiv{} & \texttt{lambda}\{\}(x.b) \end{array}$$

The meaning of sentences is given by specifying which sentences are *true*, and which sentences can be derived from other true sentences. The specification of these *judgments* uses a meta-notation involving explicit substitution and extended Horn clauses. For example the rule for and-introduction in a sequent calculus, normally written

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \text{and\_intro}$$

would be specified as the clause

$$(\Gamma \vdash A) \Rightarrow (\Gamma \vdash B) \Rightarrow (\Gamma \vdash A \wedge B),$$

where the $\Rightarrow$ is the Horn implication. Substitution is specified with second-order matching variables of the form $t[v_1, \ldots, v_n]$, which specifies terms with possible free occurrences of the variables $v_1, \ldots, v_n$, and second-order instances

$t[t_1, \ldots, t_n]$, which specifies the term t with $t_1, \ldots, t_n$ simultaneously substituted for $v_1, \ldots, v_n$. In addition, judgments may specify contexts (terms $C[]$ that contain a "hole"). For instance, the judgment for beta-reduction might appear as follows.

$$\Gamma \vdash C[(\lambda x.M[x])\ N] \Rightarrow \Gamma \vdash C[M[N]]$$

Let *term*$^+$ denote terms extended with second-order variables and contexts, then the general syntax for a judgment is as follows.

$$\boxed{\begin{array}{rl} \mathit{judgement} ::= & \mathit{term}^+ \\ \mid & \mathit{term}^+ \Rightarrow \mathit{judgement} \end{array}}$$

Mathematical theories are formulated as *theories*, which are an extension of the ML module system. The syntax for theories is shown in Figure 1. We discuss the meaning of these statements in the next few paragraphs.

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

Fig. 1. Theory syntax

`axiom, prim, thm` A signature declares the rules and structure of a theory, without giving it an implementation, and a theory implementation provides proofs for the declarations. A rule is declared in a theory with an `axiom` form. The declaration

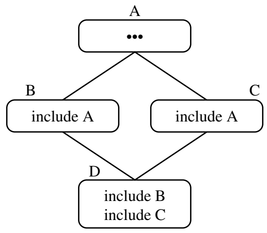
$$\texttt{axiom}\ \mathit{name}\ p_1 \ldots p_n : t_1 \Rightarrow \cdots \Rightarrow t_m$$

specifies that the term t_m is true, if the antecedents $t_1, \ldots, t_{m-1}$ are true. This axiom declares the inference rule, but it does not justify it. To justify an `axiom`, the implementation may specify that the axiom is true by *assumption*, with the `prim` form, or it may derive it from other rules as a `thm`. 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 `prim` form is used only when the type theory is defined. Afterwards, rules are justified as theorems, using the `thm` form, which provides a *tactic* to prove the specified goal. A *tactic* is a packaged rule for backward chaining that includes a method for finding the justification of a rule from an implementation. When an `axiom` is declared, the framework generates a *tactic* that is used for *refining* a goal by backward chaining. The `prim` and `thm` forms also declare tactics, all of which are bound as ML values of the same name. All of the forms also take additional term arguments $p_1, \ldots, p_n$ that are used to specify extra arguments that may be needed for backward chaining. For instance, a declaration of the "cut" rule would require two extra arguments, one for new assertion, and another for its name.

$$\texttt{prim}\ \mathit{cut}\ x\ A : (a\colon \Gamma \vdash A)\ (b\colon \Gamma, x\colon A \vdash C) = (b[a/x] : \Gamma \vdash C)$$



$\boxed{\texttt{include}}$ 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. Syntactically, a theory declares itself to be a subclass of another class by using the "`include` *name*," where *name* is the name of a theory signature. 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.

In a theory implementation, an `include` directive specifies that another implementation should be inlined. As before, multiple implementations are suppressed.

$\boxed{\textit{Other forms}}$ 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. The `rewrite` form is really a derived form—it would also be possible to declare rewrites as rules that allow the rewrite in any context:

$$\texttt{axiom}\ \mathit{beta} : C[(\lambda x.M[x])\ N] \Rightarrow C[M[N]].$$

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. For instance, addition in a number theory is different from addition in real analysis. Each term also has a longer name prefixed by the name of the module, so that, for example, a module using both number theory and real analysis can access both forms of addition.

Definitions are a combination of a new term declaration, and a primitive rewrite that gives a meaning to the new term. For instance, the predicate for positive numbers might be defined as follows:

$$\begin{array}{l}\texttt{declare positive}\{\}(i)\\ \texttt{primrw rw\_positive : positive}\{\}(i) \Longleftrightarrow (i > 0).\end{array}$$

2.2 Rules and Tactics

So far, we haven't spoken much about how theorems can be derived. The primitive formal framework contains a term language and allows the declaration of judgments. The primitive meta-logic contains a single rule, which is a restricted version of modus-ponens in a meta-sequent calculus. We will denote these meta-sequents with the turnstile $\models$ to distinguish them from sequents in the term language. The primitive inference rule allows a goal to be derived if the assumptions contain an assumed judgment that specifies the goal as its result, and all the antecedents of the judgment are derivable.

$$\frac{\Gamma, r{:}\, T_1 \Rightarrow \cdots \Rightarrow T_n, \Delta \models T_i \quad i \in \{1 \ldots n-1\}}{\Gamma, r{:}\, T_1 \Rightarrow \cdots \Rightarrow T_n, \Delta \models T_n}$$

Conceptually, during a proof, the assumption list contains all rules that have been previously declared in the theory containing the proof–including any rules declared in parent theories. Each `axiom` statement declares an infinite number of rules, one for each substitution instance. For instance, the statement `axiom` *hyp* : $\Gamma, x{:}\, A, \Delta \vdash A$ declares a rule for any variable for x, any term for A, and any term sequences for Γ and Δ.

As we stated in Section 2.1, when an axiom is declared, the framework automatically produces a tactic, which provides a handle that can be used request a primitive application of the rule. The tactic for and-introduction, when applied, would compute two subgoals, one for each branch of the conjunction. Tactics are associated with the theories in which they are declared, and the framework enforces the restriction by allowing the tactic to be applied only in proofs in sub-theories.

Tactics can also be combined with a few combinators (also called *tacticals*). The `andthen : tactic -> tactic -> tactic` tactical applies its first tactic, and then applies the second tactic to all the subgoals. The `orelse : tactic -> tactic -> tactic` applies the first tactic, and if it fails by raising a refinement exception, applies the second tactic. The `thenL : tactic -> tactic list -> tactic` tactical applies the first tactic, and then maps the tactic list across the subgoals, which must have the same number.

2.3 Formal Theories

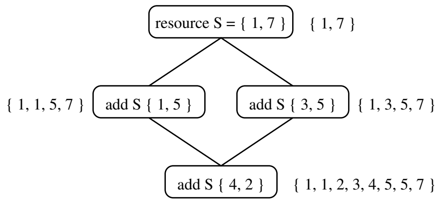
One of the key features of the framework is that theories and their signatures are first class. The framework provides a means to extract a formal type from a theory signature, and a formal object from its implementation. The general idea is to translate a module signature to a dependent record type, 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 [13].

2.4 Proof Automation

Because of the undecidability of higher-order logics, higher order theorem provers are typically designed to be interactive. The goals are presented to the user and interaction is typically by "refinement" (backward-chaining). However, the vast majority of steps in a formal proof are trivial, and a great deal of effort is exerted to automate the "obvious" steps in the proof with decision procedures or heuristics.

In a modular system, the task becomes more difficult because of the potential for multiple distinct logics. In many cases, proofs occur in type-theory fragments that may not include a complete collection of type constructors. In fact, the situation potentially becomes even more difficult. As rules are added to a logic, the decision procedures may change drastically (consider the case where the law of excluded middle is added intuitionistic propositional logic). We do not address the algorithmic changes here. Instead, we address the issue of modular proof automation with a device called *resources*.

Intuitively, a *resource* is a property of a type theory that can be produced by combining the resources of its parts. Syntactically, a resource is like a method that automatically includes the values of the methods it is inheriting. For example, the resource S in the diagram in the next paragraph computes the union of the multisets in each module. Note that the multiset in the root theory is inherited only once.



From the view of the implementation, a resource is really composed of three methods, `create`, `add`, and `join`. The `create` method creates an empty resource in a root theory. The `add` method adds a value to an inherited resource (in the diagram, the statement `add` S $\{1,5\}$ computes the union of $\{1,5\}$ and $\{1,7\}$ and assigns it to the resource). The `join` method combines the value of a resource that is inherited from multiple ancestors (the final class computes the `join` of the sets $\{1,1,5,7\}$ and $\{1,3,5,6\}$ before the `add` S $\{4,2\}$ statement).

Typically, resources are used to collect the parts of a modular reasoning method. For instance, a type theory may implement modules for each of its type constructors, and it might provide a decision procedure that has a component for each type constructor. These components would be collected in a resource, from which a tactic would ultimately be extracted. An example of such a resource is given in the first example, where a decision procedure is generated for the intuitionistic propositional logic.

3 Examples

The following example illustrate three properties of the framework. The first example is a basic specification of a fragment of the intuitionistic propositional logic (IPL). The second example is intended to illustrate how logics can be related, by providing an interpretation of IPL in the Nuprl type theory (ITT). The final example illustrates the first–class object–oriented modules by developing the canonical one–dimensional point example. Due to space limitations, these examples are quite brief. However, we have developed significant examples, including a specification of LF and the Nuprl type theory in the framework. More examples can be found at the WWW site [12].

3.1 Intuitionistic Propositional Logic (IPL)

The first example defines an intuitionistic propositional logic with only implication and falsehood. This a simple example, but it covers many of steps in defining a logic, including the construction of a modular decision procedure. The logic is formalized with sequents, declared in the root theory `ipl_root_sig`. The root signature also declares the resource `prove` that is used to construct the decision procedure for the logic. The next two signatures are for the type constructors $\perp$ and $\Rightarrow$. Each module `declare`s the syntax for the type constructor, as well as any definitions and inference rules. For instance, the signature `ipl_implies_sig` defines a primitive term for implication, and defines negation in terms of implication and falsehood.

```
theory_sig ipl_root_sig =
    declare Γ ⊢ C
    axiom assume : Γ, x: A, Δ ⊢ A
    resource prove : tactic → tactic
end

theory_sig ipl_false_sig =
    include ipl_root_sig
    declare ⊥
    declare any
    axiom false_elim : Γ, x: ⊥, Δ ⊢ C
end

theory_sig ipl_implies_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

theory_sig ipl_sig =
    include ipl_false_sig
    include ipl_implies_sig
    val decide : tactic
    axiom example : Γ ⊢ (A ⇒ B) ⇒ (A ⇒ ¬B) ⇒ ¬A
end
```

Fig. 2. Modular signature of propositional logic

The implementation `ipl_root` implements the primitive resource, and also provides the primitive proof extract for the `assume` rule. Each implementation provides a primitive proof extract, as well as a part of the `prove` decision procedure specific to the module in question. In the final theory, the algorithm is extracted and assigned to the tactic `decide`. This particular proof algorithm is not complete, but a more complete implementation would give a decision procedure.

In this example, we make use of additional tacticals that are coded out of the primitive tacticals. These include `onsomehyp`, which takes a tactic and applies it to each hypothesis until a proof is found (if there is none, it fails). The tactical `oneof : (tactic -> tactic) list -> tactic -> tactic`, tries each tactical in the argument list, applying it to the tactic argument until a proof

```
theory ipl_root =
    let prove = {
        create = [];
        add = function x l → x::l;
        join = function l1 l2 → l1 @ l2;
        extract = function l → let rec prove p = oneof l prove p in prove
    }
    prim assum = x : (Γ, x: A, Δ ⊢ A)
    add prove (fun _ → onsomehyp assum)
end

theory ipl_false =
    include ipl_root
    prim false_elim = any : (Γ, x: ⊥, Δ ⊢ C)
    add prove (fun _ → onsomehyp false_elim)
end

theory ipl_implies =
    include ipl_false
    primrw neg_rw : ¬A ⟺ (A ⇒ ⊥)
    prim imp_intro a : b[a]: (Γ, a: A ⊢ B) = λa.b[a] : (Γ ⊢ A → B)
    prim imp_elim : (a: (Γ, Δ ⊢ A)) (b[x]: (Γ, x: B, Δ ⊢ C)) = b[a] : (Γ, x: A ⇒ B, Δ ⊢ C)
    add prove (fun t → (imp_intro andthen t) orelse ((onsomehyp imp_elim) andthen t))
end

theory ipl =
    include ipl_implies
    include ipl_true
    let decide = prove.extract
    thm example = decide : Γ ⊢ (A ⇒ B) ⇒ (A ⇒ ¬B) ⇒ ¬A
end
```

Fig. 3. Implementation of propositional logic

is found. The proof algorithm is collected as a list of tacticals to try. When the proof algorithm is extracted, it saves a copy of the saved tactical list and proof search is guided by the `prove` function.

3.2 Relating IPL to Nuprl Type Theory

The propositional logic specification just explored is axiomatic. The syntax and rules for the logic are states, and the implementations provide primitive proof extracts for each of the rules. 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 Figure 4, we show how to derive an implementation fro `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 : $(A \Rightarrow B) \Longleftrightarrow \texttt{implies}(A, B)$
    rewrite lam_def : $\lambda x.b[x] \Longleftrightarrow \texttt{lambda}(x.b[x])$
    rewrite apply_def : $(M\ N) \Longleftrightarrow \texttt{apply}(M; N)$

    thm imp_intro a $(\Gamma, a{:}\, A \vdash B)$ =
        (unfold imp_def andthen itt_imp_intro) : $(\Gamma \vdash A \rightarrow B)$

    thm imp_elim $(\Gamma, \Delta \vdash A)$ $(\Gamma, x{:}\, B, \Delta \vdash C)$ =
        (unfold imp_def andthen itt_imp_elim) : $(\Gamma, x{:}\, A \Rightarrow B, \Delta \vdash C)$

    add prove ...
end

Fig. 4. Derivation of IPL implication from Nuprl

3.3 Point Objects

For the final example, we illustrate some of the object-oriented features of the formal system by specifying one-dimensional movable points in the Nuprl type theory. A point has a location, and two methods: the `getX` method returns the location, and the `bumpX` method creates a new point with a shifted location. In addition, the `Point` object contains the specification that the `getX` and `bumpX` have the correct behavior. In this example, we give only the signatures (although implementation can be shown to exist). This example follows an encoding of object similar to the existential interpretation of Pierce and Turner [27], with the exception that the "state" or "carrier" of the object (`car`) is not abstract. A deeper encoding would use existential types to hide the value of the `car` method. More detail of this interpretation is given in Hickey [13].

In this example, the `bumpX` method must be polymorphic over subobjects. Subobjects are specified with the $\preceq$ relation, part of the type theory, and the polymorphism is expressed using an intersection type, quantified over all subobjects. The `ColorPoint` signature `include`s the `Point` signatures, and as a result, the `Point` methods are inherited by the `ColorPoint`. It can be shown that the type `ColorPoint` is a subtype of `Point`, i.e. `ColorPoint` $\preceq$ `Point`.

```
theory_sig Point =
    include ITT
    axiom car : ⊢ Type
    axiom zero : ⊢ car
    axiom getX : ⊢ car → Z
    axiom bumpX : ⊢ ⋂_{T≼car} .(T → Z → T)
    axiom spec : ⊢ ∀x: car.∀i: Z.getX(bumpX x i) = getX(x) + i
end

theory_sig ColorPoint =
    include Point
    axiom getC : ⊢ car → color
    axiom setC : ⊢ ⋂_{T≼car} .(T → color → T)
end

theory_sig PointWrapper =
    include ITT
    axiom zero : ⊢ Point
    axiom getX : ⊢ Point → Z
    axiom bumpX : ⊢ ⋂_{P≼Point} .(P → Z → P)
    axiom spec : ⊢ ∀p: Point.∀i: Z.getX(bumpX p i) = getX(p) + i
end

theory pointWrapper =
    include ITT
    thm getX p = p.getX p.car
    thm bumpX p i = p.car ← (p.setX p.car i)
    ⋮
end
```

Fig. 5. One dimensional movable points

In the final signature, `PointWrapper`, the interface methods of the `Point` are defined to operate directly on objects of type `Point`. In the implementation `pointWrapper` of this module, the `getX` method projects the value in `car`, and applies the primitive `getX` method. The `bumpX` method operates similarly, but repackages the object for the result value. We use loose notation in this implementation—the method values listed are actually the proof extracts of the proofs of the method type, and method construction follows the propositions-as-types principle.

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.

5 Related Work

Our framework draws on the work of Jackson [18, 17], who formulated a great deal of abstract constructive algebra in the Nuprl system. Jackson's system formalized algebraic objects in the type theory using dependent Cartesian products, which suffered from the lack of convenient subtyping properties. Our development originally began as a means of addressing the problem of higher-order dependent modules with the expected subtyping properties (a module with more items is a subtype of a module with fewer). This led to the use of object–oriented techniques where higher order modules are formalized as objects, and the expected subtyping properties are fulfilled by object subsumption. Our object interpretation in this paper is closely modeled on the interpretation of Hofmann, Pierce, and Turner [14, 27], where objects are abstracted over a "state" type. We also draw on the interpretation of Abadi, Cardelli, and Viswanathan [1, 2], where objects contain only methods and state update is provided by method override.

There has been a great deal of research on logical frameworks. Our work develops the meta-logical framework (in Nuprl) of Constable and Basin [4]. Nuprl itself was developed out of the early work of Martin–Löf [20, 21], as well as the Automath logical framework [22]. Our framework has much in common with the Isabelle generic theorem prover [23, 25], which is based on hereditary Harrop formulas. Both *Nuprl-Light* and Isabelle provide a theorem prover framework that can be used to formalize generic logics, and in practice the logical specification is quite similar. A difference of the two is that logics in *Nuprl-Light* are intended to be *related.* If, for instance, an interpretation of type theory is available in set theory, the interpretation should be formalizable.

We can also compare *Nuprl-Light* to the (Edinburgh) Logical Framework [9] (which is implemented in ELF [26], for instance). In a sense, our framework has a different purpose than LF—where LF provides a framework for logics, our framework is for their *implementations.* Our basic logic is much weaker, and type theories, including LF, are implemented by asserting their inference rules, much like Isabelle. Harper and Pfenning [11] propose a module system for LF, which is similar to ours in some ways. However, we place an additional emphasis on *relations* between logics through object–oriented techniques.

On this theme, Martí-Oliet and Meseguér [19] propose rewriting logics a solution to the proliferation of logics; they also propose object–oriented theories. Guinchiglia et. al. [6] are also exploring a general architecture where provers can be combined in a "plug–and–play" manner. As this task proceeds, we need a *semantic* basis for relating theories, as Howe [16] provides in his semantics for HOL [7] and Nuprl.

6 Conclusion

We have described the *Nuprl-Light* framework, which extends the results of generic theorem provers by adding formal, first–class theories. First–class theories enable a new style of reasoning where multiple type theories and theorem provers participate in large scale reasoning. By allowing type theories to be related formally, provers may use multiple mathematical domains for their proofs, relying on the framework to construct the foundational justification.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2. Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An Interpretation of Objects and Object Types. In *ACM Symposium on Principles of Programming Languages*, 1996.
3. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and Willaim Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Conference on Logic in Computer Science*, pages 195–197, June 1987.
4. David A. Basin and Robert L. Constable. *Metalogical Frameworks*, pages 1–29. Cambridge University Press, 1993.
5. R.L. Constable et.al. *Implementing Mathematics in the NuPRL Proof Development System*. Prentice–Hall, 1986.
6. F. Giunchiglia, P. Pecchiari, and C. Talcott. Reasoning Theories: Towards an Architecture for Open Mechanized Reasoning Systems. Technical Report 9409-15, IRST, Trento, Italy, 1994.
7. M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
8. Timothy G. Griffin. *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems*. PhD thesis, Cornell University, 1988.
9. Rober Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1), January 1993.
10. 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.

11. Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, To appear. A preliminary version is available as Technical Report CMU-CS-92-191.
12. Jason J. Hickey. Nuprl-light. `http://www.cs.cornell.edu/home/jyh`.
13. 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.
14. Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995.
15. Douglas J. Howe. Equality in Lazy Computation Systems. In *Fourth Annual Symposium on Logic in Computer Science*, pages 198–203. IEEE Computyer Society Press, 1989.
16. Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In *AMAST '96*, 1996.
17. Paul Jackson. Exploring Abstract Algebra in Constructive Type Theory. In *12th International Conference on Automated Deduction*, pages 590–604. Springer, 1994.
18. Paul Bernard Jackson. *Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, January 1995.
19. Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, 1993.
20. Per Martin–Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North–Holland, 1975.
21. Per Martin–Löf. *Intutionistic Type Theory.* Bibliopolis, Napoli, 1984.
22. R.P. Nederpelt, J.H. Geuvers, and editors R.C.de Vrijer. *Selected Papers on Automath*, volume 133. North-Holland, 1994.
23. Tobias Nipkow and Lawrence C. Paulson. Isabelle-91. In *Proceedings of the 11th International Conference on Automated Deduction*, pages 673–676, 1992.
24. Lawrence C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF.* Cambridge Univ. Press, 1987.
25. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer LNCS 828, 1994.
26. Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815. Springer LNAI 814, June 1994.
27. Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

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