MetaPRL – A Modular Logical Environment*

Jason Hickey[1], Aleksey Nogin[1], Robert L. Constable[2], Brian E. Aydemir[1], Eli Barzilay[2], Yegor Bryukhov[3], Richard Eaton[2], Adam Granicz[1], Alexei Kopylov[2], Christoph Kreitz[2], Vladimir N. Krupski[4], Lori Lorigo[2], Stephan Schmitt[5], Carl Witty[6], and Xin Yu[1]

[1] Department of Computer Science, California Institute of Technology
M/C 256-80, Pasadena, CA 91125
`{jyh,nogin,emre,granicz,xiny}@cs.caltech.edu`

[2] Department of Computer Science, Cornell University,
Ithaca, NY 14853
`{rc,eli,eaton,kopylov,kreitz,lolorigo}@cs.cornell.edu`

[3] Graduate Center, City University of New York
365 Fifth Avenue, New York, NY 10016
`ybryukhov@gc.cuny.edu`

[4] Laboratory for Logical Problems of Computer Science
Department of Mathematical Logic and Theory of Algorithms
Faculty of Mechanics and Mathematics, Moscow State University
Vorob'evy Gory, 119899 RUSSIA
`krupski@lpcs.math.msu.ru`

[5] Sapient, Presidency Building,
Mehrauli-Gurgaon Road, Sector-14,
Gurgaon-112 001, Haryana, India
`sschmitt@sapient.com`

[6] Newton Research Labs,
441 SW 41st Street,
Renton, WA 98055
`cwitty@newtonlabs.com`

Abstract. MetaPRL is the latest system to come out of over twenty five years of research by the Cornell PRL group. While initially created at Cornell, MetaPRL is currently a collaborative project involving several universities in several countries. The MetaPRL system combines the properties of an interactive LCF-style tactic-based proof assistant, a logical framework, a logical programming environment, and a formal methods programming toolkit. MetaPRL is distributed under an open-source license and can be downloaded from `http://metaprl.org/`. This paper provides an overview of the system focusing on the features that did not exist in the previous generations of PRL systems.

* This work was supported in part by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research (ONR) under Grant N00014-01-1-0765, the Defense Advanced Research Projects Agency (DARPA), the United States Air Force, the Lee Center, and by NSF Grant CCR 0204193.

1 Introduction

MetaPRL is the latest in the PRL family of systems [5,11,12,17,18,29,49] developed over the last 25 years. MetaPRL's predecessor NuPRL [5,18] was successfully used for verification and automated optimization of the Ensemble group communication toolkit [14,38]. The Ensemble toolkit [23] is being used for both military and commercial applications; its users include BBN, Nortel Networks and NASA.

The MetaPRL project (which was initially called NuPRL-Light [27]) was started by Jason Hickey as a part of Ensemble verification effort to simplify formal reasoning about the program code and to address scalability and modularity limitations of NuPRL-4. As more effort was put into the system, MetaPRL eventually grew into a very general modern system whose modularity on all levels gives it flexibility to support a very wide range of applications.

MetaPRL is not only a tactic-based interactive proof assistant, it is also *a logical framework* that allows users to specify their own logical theories rather than requiring them to use a single theory. Additionally, MetaPRL is a logical programming environment that incorporates many features to simplify reasoning about programs being developed. In fact, MetaPRL is implemented as an extension of the OCaml compiler [50]. Finally, MetaPRL can be considered *a logical toolkit* that exports not only the "high-level" logical interface, but all the intermediary ones as well. This allows for rapid development of applications that require formal or semi-formal handling of data.

While MetaPRL was written from scratch and without using any of the preexisting PRL code, it keeps many of the major design principles and concepts of the NuPRL system. For example, the two systems have very similar term syntax and MetaPRL implements several variations of the NuPRL type theory as one of its logics (see Section 5.2).

However, MetaPRL is substantially different from NuPRL and has many new features. In this paper we present an overview of the system focusing on the features that were introduced in MetaPRL and that did not exist in previous generations of PRL systems.

MetaPRL is an open-source software system distributed under the terms of the GNU GPL. Documentation and download instructions can be found at [32].

2 Architecture Overview

At a very high level, the architecture of a tactic-based theorem prover can usually be described as a layered architecture as shown in Figure 1.

The core of the system is its *logical engine*, or *refiner* [10]. It is responsible for performing the individual proof steps (such as applying a single inference rule). Next, there is the lower "support" layer for the logical theories. It usually includes basic meta-theory definitions and possibly some basic proof search mechanisms (such as basic tactics). Finally, at the top of the structure there are the logical theories themselves, each potentially equipped with theory-specific mechanisms (such as theory-specific proof search strategies and theory-specific display mechanisms). In a way, the structure of the prover mimics the structure of an operating system with logical engine being the "kernel" of the system, meta-theory being its "system library" and logical theories being its "user space".

We intentionally did not include any user interface in Figure 1. The reason for such omission is that often a user interface (such as the NuPRL Editor [5,39] or Proof General [6]) would be a separate package added on top of a formal system, rather than a part of the system itself.

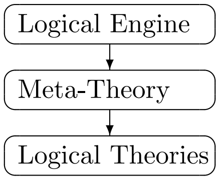

Fig. 1

There are two main approaches to building such a prover — one can build a monolithic prover (such as NuPRL-4) or one can build a modular one. There are several advantages in a more modular architecture, especially in a research environment where we want to work on general methodology of formal reasoning.

In a modular system with well-defined interfaces it is easier to try out new ideas and new approaches. This allows for a greater flexibility and also helps in bringing new people (including new students) to the project.

The modular architecture also allows one to have several implementations of some critical module. For example, it is possible to have a generic implementation and at the same time create alternative implementations of some modules that are optimized towards a particular class of applications. This approach is especially useful in the trusted core of the system — there we can have a simple "reference" implementation that is extensively tested and checked for correctness as well as one (or more[1]) highly optimized implementations. Users can develop proofs using the optimized modules and then later double-check them by re-running the proof scripts using the reference implementation. This provides the confidence of knowing that proofs were accepted by *both* implementations.

Similarly to the modularity of the logical engine of a formal system, the modularity of the logical theories supported by a system is also important. Some provers only support reasoning in a single monolithic logical theory, while others, including MetaPRL, not only give their users a choice of which logical theory to use, but also allow users to add their own logical theories to the system. Such systems are often called *logical frameworks* [47].

MetaPRL provides an implementation of the architecture presented in Figure 1. The implementation is highly modular on all levels — from logical engine to logical theories.

The structure of the paper follows the structure of the system. In Section 3 we present the features of the MetaPRL logical engine, in Section 4 we present the features of MetaPRL intermediate layer, and in Section 5 we present an overview of logical theories in MetaPRL. We present the logical toolkit side of the system in Section 6 and provide a brief overview of the related work in Section 7.

[1] In fact, in MetaPRL some of the most performance-sensitive modules have up to 6 different implementations.

3 Logical Engine

The core of the system is its *logical engine* or *refiner* [10] that performs two basic operations. First, it builds the basic proof procedures from the parts of a logic. The second refiner operation is the *application* of the basic proof steps producing justifications from the proofs.

The MetaPRL refiner is based on a higher-order term rewriting engine. This rewriting engine is used to apply the rules of the system (including both the axioms and the derived rules described in Section 3.3) by rewriting the current proof goal term into terms representing the subgoals that remain to be proven. The rewriting engine is also used to apply computational and definitional rewrites (see Section 3.4). When a rule or rewrite is defined in a logical theory, the MetaPRL refiner compiles it to a bytecode program [31] that is run whenever the rule or rewrite is applied. This precompilation phase significantly improves performance.

The rewriting engine also has an "informal" mode that is used to convert terms into strings to be displayed to a user (or to be written into a LaTeX file). This informal mode is also used to provide generic parsing capabilities and enables users to specify parts of their logical theories in their own notation [21]. The rewriting engine is used to execute parsing derivations based on the formal definition of the notation, which includes the specification of the grammar and the semantic rules associated with each grammar production. For instance, one can define a logical theory to reason about simple functional programs and use actual programming syntax in rewrite rules to specify formal transformations. When experimental parsing capabilities are more tightly integrated into the system, the definitions of the notation will become an integral part of the logical theories making the logical content more apparent and easy to understand.

3.1 Speed

In a tactic-based prover, the speed of the underlying logical engine has a direct impact on the level of reasoning. If proof search is slow, more interactive user guidance is needed to prune the search space, leading to excessive detail in the tactic proofs. And if the system is fast, it allows users to concentrate more on the high-level reasoning leaving it to the machine to fill in the "trivial" details.

MetaPRL was designed with efficiency in mind. In addition, MetaPRL code is highly modular, which has made it easy to improve the efficiency of the procedures along the critical path (the rewriting engine). MetaPRL modularity has also allowed us to replace generic modules with domain-specific implementations that improve performance in some logics. As we explained in Section 2, adding complex optimizations even to the "trusted core" of the system does not increase the potential exposure to bugs since the proofs developed using the optimized refiner can still be double-checked using the slower more trusted implementation.

As a result of our speed-conscious design and implementation (described in detail in [31]) as well as the quality of the OCaml compiler, the MetaPRL logical engine is considerably faster than NuPRL-4. We compared the two systems by writing tactics that implement a simple domain-specific proof search algorithm in each of the systems. We performed several tests in several domains and in all cases MetaPRL was over 100 times faster. And by distributing the system over several processors and several computers we were able to achieve even greater speed-ups.

3.2 Transparent Concurrent and Distributed Refinement

MetaPRL is capable of distributing a proof search over several processors using the Ensemble group communication system [23]. The distribution is transparent for both the tactic programmer and the system user. That is, the tactics are programmed using a language very similar to that of NuPRL without restriction. Processes may join and leave (even fail) at any time, affecting only the speed of the distributed proof search. On a small number of processors, speed improvements are usually superlinear in the number of processors participating in a proof.

The distribution mechanism is described in-depth in [28].

3.3 Derived Rules

In an interactive theorem prover it is very useful to have a mechanism allowing users to prove some statement in advance and then reuse the derivation in further proofs. Often it is especially useful to be able to *abstract* the particular derivation. For example, suppose we wish to formalize a data structure for labeled binary trees. If binary trees are not primitive to the system, we might implement them in several ways, but the details are irrelevant. The more important feature is the inference rule for induction. In a sequent logic, the induction principle would be similar to the following: for an arbitrary predicate P,

$$\frac{\Gamma \vdash P(\mathit{leaf}) \qquad \Gamma, a\colon \mathit{btree}, P(a), b\colon \mathit{btree}, P(b) \vdash P(\mathit{node}(a,b))}{\Gamma, x\colon \mathit{btree} \vdash P(x)}$$

If this rule can be established, further proofs may use it to reason about binary trees *abstractly* without having to unfold the *btree* definition. This leaves the user free to replace or augment the implementation of binary trees as long as she can still prove the same induction principle for the new implementation. Furthermore, in predicative logics, or in cases where well-formedness is defined logically, the inference rule is strictly more powerful than its propositional form. For example, in NuPRL-style type theories certain induction principles can only be expressed as rules and can not be fully expressed in a propositional form.

If a mechanism for establishing a derived rule is not available, one alternative is to construct a proof "script" or tactic that can be reapplied whenever a derivation is needed. There are several problems with this. First, it is inefficient — instead of applying the derived rule in a single step, the system has to run through the entire proof each time. Second, the proof script would have to unfold the *btree* definition, exposing implementation detail. Third, proof scripts tend to be fragile, and must be reconstructed frequently as a system evolves. Finally, by looking at a proof script or a tactic code, it may be hard to see what exactly it does, while a derived rule is essentially self-documenting.

Another advantage of derived rules is that they usually contain some information on how they are supposed to be used. For example, an implication $A \Rightarrow B$ can be stated and proved as an A elimination rule or as a B introduction rule, depending on how we expect it to be used. As we will see in Section 4.2, such information can be made available to the proof automation procedures, significantly reducing the amount of information users have to provide manually.

MetaPRL provides a purely syntactical mechanism for *derived rules.* The mechanism is very general and does not depend on a particular logical theory being used. The key idea of our approach is in using a special higher-order language for specifying rules; we call it a *sequent schemata* language [44]. From a theoretical point of view, we first take some logical theory and express its rules using sequent schemata. Next, we extend the language of the theory with the language of sequent schemata. After that we allow extending our theory with a new *derived rule* $\frac{S_1 \ \cdots \ S_n}{S}$ whenever we can prove S from S_i in the expanded theory. We have shown [44] that this mechanism would only allow deriving statements that were already derivable in a conservative extension of the original theory.

In MetaPRL the user only has to provide the axioms of the base theory in a sequent schemata language and the rest happens automatically. The system immediately allows the user to mix the object language of a theory with the sequent schemata meta-language. Whenever a derived rule is proven in a system, it allows using that rule in further proofs as if it were a basic postulate of the theory.[2]

3.4 Computational Rewrites

In MetaPRL it is possible to define not only logical rules, but also logical rewrites. A logical rewrite states an equivalence between two terms is valid in any context. For example, in NuPRL-style type theory, the computationally equivalent terms, such as $\lambda x.A(x)\ B$ and $A(B)$, can always be interchanged.

MetaPRL also allows "rewrite theorems" (derived rewrites) and conditional rewrites — rewrites that state that two terms can be interchanged in contexts where a certain condition is true. For example, the rewrite $(x \neq 0) \longrightarrow (x/x \longleftrightarrow 1)$ states that in any context where x is known to be non-zero, x/x can be interchanged with 1.

This powerful rewrite mechanism allows MetaPRL users to avoid stating and proving well-formedness subgoals in cases when they are not really necessary.

[2] MetaPRL would also allow the reverse order — first state a derived rule, use it, and later "come back" and prove the derived rule. Of course, this means that a proof is not considered complete until all the derived rules used in it are also proven. Such an approach allows one to "test-drive" a derived rule before investing time into establishing its admissibility.

Additionally, the context-independence of rewrites enables us to chain rewrite applications (and rewrite application attempts) in a very efficient manner, making rewrite applications an order of magnitude faster than rule applications.

4 Proof Search Automation

In addition to the logical engine, MetaPRL also provides considerable proof automation, using extensible proof-search procedures coded as LCF-style [20] *tactics*.

4.1 Resources

Often some basic tactics are designed to behave very differently in different contexts. One of the best examples of such a tactic is the *decomposition tactic* [33, Section 3.3] present both in NuPRL and in MetaPRL. When applied to the conclusion of a goal sequent, it will try to decompose the conclusion into simpler ones, normally by using an appropriate introduction rule. When applied to a hypothesis, the decomposition tactic would try to break the hypothesis into simpler ones, usually by applying an appropriate elimination rule.

Even with a fixed base logic, as in NuPRL, these automated procedures need to be updated dynamically as new definitions and theorems are added. In MetaPRL, with multiple (perhaps conflicting) logics, this has the added complexity that definitions and theorems can be used for automation only in the logic in which they are defined or proved.

MetaPRL automates this process through a mechanism called *resources*. A resource is essentially a collection of pieces of data (with each piece of data residing in a particular logical theory); the resource interface provides an *inheritance* mechanism based on the logical hierarchy (see Section 5.1). Resources are managed on a per-theorem granularity — when working on a particular proof, the resource state reflects everything collected from the current theory up to the theorem being proved, as well as everything inherited from the theories that are ancestors of the current one in the logical hierarchy.

MetaPRL has resources controlling the behavior of the decomposition tactic, of the type inference heuristic, of the term simplifier rewriting tactic, and many others.

4.2 Resource Annotations

When a new rule (or rewrite) is added to a system, new data has to be added to some resources if we want to allow the proof search procedures controlled by those resources to take advantage of the new rule (rewrite). It turns out that most such resource updates are rather uniform. For most MetaPRL resources we have been able to automate these resource insertions by giving the resource updating functions access to the *text* of the newly added rules and rewrites (as opposed to just giving them access to primitive tactics corresponding to those rules and rewrites), essentially creating a *reflective* mechanism. This is possible because all rules and rewrites are expressed in a formally defined language of sequent schemata (see Section 3.3).

From the MetaPRL user's perspective this mechanism has a form of *resource annotations*. When adding a new rule, a user only has to annotate it with the names of resources that need to be automatically improved. Users can also pass some optional arguments to the automatic procedure in order to modify its behavior. As a result, when a new logical object (rule, rewrite, etc) is added to a MetaPRL theory, the user can usually update all relevant proof search automation by only typing a few extra symbols. Moreover, adding new resources is quite easy, and there are many tools that make automation of resource improvements simpler.

For more information on resource annotations, see [43, Section 4.3].

4.3 Generic Tactics

Derived rules and resource annotations combined provide a new way of implementing many complex tactics. Instead of writing large tactics code that may be hard to debug and to understand, MetaPRL users can view a tactic as a number of deterministic sequences of rule applications together with some control information that specifies which sequences get executed and in what order. Deterministic sequences would be implemented as derived rules, and control information would be added as resource annotations on some of the rules. This improves the efficiency of these tactics (applying a derived rule only takes one step of the rewriting engine) and usually makes them easier to maintain.

When a tactic is implemented via resource annotations, most of its code is *generic* and does not depend on particular details of a logical theory. The great advantage of such generic tactics is that they can be implemented once and then reused in a wide range of logical theories with no or a little additional effort. In a logical framework like MetaPRL this leads to a significant degree of code reuse and greatly simplifies the task of automating proof search when new theories are added to the system.

Another approach to creating generic tactics in MetaPRL is turning decision procedures and automated proving procedures into *heuristics*. We observe that proving the decision procedure is correct *in a particular instance* is much easier than proving that it will *always* be correct and the former can often be established automatically. When a decision procedure can be enhanced to output some evidence along with the "yes" answer, it can be turned into a tactic that first executes the enhanced decision procedure and then tries to interpret the provided evidence, turning it into a complete proof. Since tactics go through the logical engine, we now get a decision procedure that does not have to be trusted. This decouples the procedure from the theory it is being used in since we no longer have to keep making sure the procedure correctly matches the theory every time we want to change either the theory, or the decision procedure.

This approach was used by Stephan Schmitt for implementing the JProver decision procedure in MetaPRL. JProver [48] is a complete[3] theorem prover for first-order intuitionistic logic that is based on a strategy called the *connection method* [13,36]. Upon success it generates a sequent proof for the proof goal [37] that may be inspected by a user.

JProver is implemented on top of MetaPRL core in a very generic way [48], using MetaPRL as a theorem proving toolkit (see Section 6) without referring to any specific logical theory. When it finds a proof, JProver outputs a simple generic encoding of the proof that can be easily converted to a tactic in, for example, type theory. Since JProver's output is converted to a tactic and is not "trusted", this allows us to use it even when not all assumptions JProver makes about the underlying logic are actually valid (as it happens in type theory).

Both approaches to generic tactics are essentially replacing a human-intensive approach with a computer-intensive one. In case of an updatable tactic we have the system itself extracting the relevant information from the text of the rules, instead of requiring users to provide it. In case of decision procedures we eliminate the need for manually establishing the validity of a procedure and instead use a computer system for post-processing proofs that come out of the procedure.

5 Logical Theories

MetaPRL *logical theories* (or simply *logics*) can contain the following kinds of objects:

(A) *Syntax definitions* define the *language* of a logic.
(B) *Inference rules* define the primitive inferences of a logic. For instance, the first-order logic contains rules like MODUS_PONENS in a sequent calculus.

$$\frac{}{\Gamma, A \vdash A}\ \text{AXIOM} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \text{MODUS\_PONENS}$$

(C) *Rewrites* (described in Section 3.4) define computational and definitional equivalences. For example, the type theory defines functions and applications with the equivalence $(\lambda x.\, b[x])\ a \longleftrightarrow b[a]$.
(D) *Theorems* contain proofs of derived inference rules.
(E) *Tactics* provide theory-specific proof search automation.

In addition to the *formal* objects enumerated above, MetaPRL theories contain *display forms* that describe how the formal syntax should be presented to the user and how to export it to LaTeX. Most theories also contain *literate comments* that are used to generate the documentation for those theories.

An extensive documentation of MetaPRL theories (generated automatically from the literate comments and updated on a regular basis) is available at [30].

[3] Since first-order logic is undecidable, JProver will not terminate if the goal cannot be proven and must be interrupted (typically by limiting the maximum proof search depth).

5.1 Hierarchical Theories Mechanism

MetaPRL does not assume any particular theory or logic and allows users to formulate and use different logics and theories. In MetaPRL logical theories are implemented as extensions of ML modules. Each theory is usually a sequence of logical objects (such as rules and definitions) and ML code (such as tactic implementations and resource code). The theories are object-oriented, in the sense that a theory specifies a class that inherits rules and implementations from other classes. All rules (including the derived rules — see Section 3.3) that are valid in a superclass are valid in a subclass.

Such a modular mechanism has many advantages. It allows formulating a new logic by composing pieces (theories) of an existing logic and adding extra theories if necessary. For example, if a user wanted to create a theory based on product types and some extra objects, she can take the product type module from the NuPRL-style type theory implemented in MetaPRL and add other modules if necessary. A nice property of our implementation is that such a user would automatically get not only all the primitive rules about the product types, but also all the theorems about them and all the tactics and resource data (see Section 4.1) needed to work with product types and all the display forms describing how to pretty print product types.

See [27] for a detailed description of the MetaPRL logical framework.

5.2 NuPRL-Style Type Theory

Some of the most powerful and challenging logical theories implemented in theorem provers are various flavors of constructive type theory. MetaPRL is not an exception — its most extensively developed and most frequently used theory is a variation of the NuPRL intuitionistic type theory [18] (which in turn is based on the Martin-Löf type theory [41]).

The MetaPRL implementation of the type theory differs from the NuPRL's one in several major aspects. The most obvious distinction is the extensive use of computational rewrites (including derived ones), derived rules and resource annotations as well as an extensive modularization of the theory.

Another big difference is MetaPRL's approach to formalizing the notion of a quotient type. In MetaPRL the traditional monolithic axiomatization of quotients is replaced by a modular set of rules for a specially chosen set of primitive operations (as described in [42] and [43, Chapter 5]). This modular formalization of quotient types turns out to be much easier to use and free of many limitations of the traditional monolithic formalization. As an illustration of the advantages of the new approach, MetaPRL includes a theory that demonstrates how the type of *collections* (that is known to be very hard to formalize using traditional quotient types) can be naturally formalized using the new primitives.

MetaPRL also includes Kopylov's theory of extensible dependent record types [34]. Record types are an important tool for programming and are essential in formalizing object-oriented calculi [1,19,26]. Dependent record types may be used to represent modules in programming languages with their specifications. Dependent record types are also used to represent algebraic structures. In most of the previous approaches, the dependent record type was treated as primitive. MetaPRL theory defines it using a new type constructor, *dependent intersection* [34]. Dependent intersection is an intersection of two types, where the second type may depend on elements of the first one. This type constructor is built by analogy to the dependent product. It turns out that the concatenation of dependent records is a dependent intersection. This observation allows us to define the record type in a very simple way. Our record type has natural subtyping properties and we are able to extend record types. Dependent intersection can also be used to define a set type. This means that dependent intersection not only adds support for dependent records, it *simplifies* the theory at the same time.

While NuPRL uses "trusted" decision procedures to implement some of its arithmetical reasoning, MetaPRL has explicit axioms with corresponding decision procedures being implemented as generic tactics (see Section 4.3).

In addition to the purely intuitionistic type theory, MetaPRL also has a theory (implemented as a module extending the standard type theory) that allows some limited form of classical reasoning [35]. While retaining most of the constructive properties, this theory allows expressing and proving a propositional analog of Markov's principle [40]. The MetaPRL and NuPRL groups continue to use purely intuitionistic reasoning for most purposes, however this experimental theory provides a promising alternative approach to managing computational meaning of constructive proofs.

5.3 Constructive Set Theory

In [2,3], Aczel introduced Constructive Zermelo-Fraenkel set theory, CZF, and formulated an embedding of CZF into the Martin-Löf's type theory [41]. Based on Aczel's work, Hickey formally embedded CZF into the MetaPRL type theory [29]. Since Aczel's CZF theory is specified explicitly by a collection of axioms, after sets and these axioms are encoded in MetaPRL's CZF module, we can use them directly without referring to the type theory.

In [51,52], Yu provided a machine-checked formalization of the basic abstract algebra on the basis of MetaPRL's CZF implementation. She started by specifying the group axioms as a collection of inference rules, defining a logic for groups. The formalization of all other concepts in abstract algebra, such as subgroups and homomorphisms, is based on this group logic. She proved some theorems of group theory constructively from these inference rules as well as the axioms of CZF in MetaPRL, and provided an example of a formalization of a concrete group, the Klein 4-group.

5.4 Other Theories

One of the goals in MetaPRL is to maintain a close connection between the formal module system and the OCaml programming language. By making the formal system an extension of OCaml, we provide a path for adding formal reasoning to applications that have been previously developed using standard software engineering methodology. This eases the burden of programming in a formal system because formal tools (for specification, verification, documentation, etc.) need only be learned when the benefits of doing so are desired. The reason is also pedantic: to learn how to program in a formal system, we can first learn how to program informally and then augment our knowledge with a foundational mathematical understanding. The final reason is a matter of bootstrapping: we would like to use MetaPRL to reason about its own implementation but we need an implementation first!

The MC theory is the first attempt at implementing a formal compiler [7]. Terms are used to formally represent the functional intermediate representation (FIR) [25] of the Mojave Compiler Collection (MCC) within MetaPRL, and rewrites are used to give the operational semantics of the FIR. Several tactics allow MetaPRL to transform FIR code through dead code elimination and inlining. Additional ML code informally translates the FIR between MCC's internal representation and the MetaPRL term language.

6 Logical Toolkit

The MetaPRL system provides a large array of efficient modules with well-defined and very generic interfaces covering various aspects of formal reasoning. The exported functionality ranges from very low-level (term syntax, alpha-equality, unification, *etc*) to very high-level (generic proof automation procedures, an ability to reason in various logical theories), and includes a very efficient term rewriting engine. This makes it very easy to use MetaPRL as a general programming toolkit for applications requiring formal methods functionality.

One example of an application developed using MetaPRL as a programming toolkit is the JProver (see Section 4.3) automated prover for first-order intuitionistic and classical logics. Once JProver was implemented[4], it *itself* became a part of the MetaPRL toolkit. As described in [48], this allowed us to integrate JProver into the MetaPRL implementation of the type theory and into the NuPRL system. Later, Guan-Shieng Huang was able to integrate[5] JProver into Coq proof assistant [8] (without any help by the members of the PRL community).

Another example is the Phobos generic parser [21] that is powered by the MetaPRL rewriting engine. Phobos is a part of a compiler [24] for a simple ML-like language, where all program transformations (all the way from parsing to x86 assembly language) are (semi-)formally specified and are executed through the MetaPRL logical engine.

MetaPRL is also being used as a part of the Formal Digital Library (FDL) project being developed at Cornell, Caltech and Wyoming. The first prototype

[4] The main JProver developer did not have any previous experience with MetaPRL.

[5] See `http://coqcvs.inria.fr/cgi-bin/cvswebcoq.cgi/checkout/V7/contrib/jprover/README` for more information on Coq JProver integration.

FDL has been built [4] and contains definitions, theorems, theories, proof methods, and articles about topics in computational mathematics and books assembled from them. Currently it supports these objects created with the theorem proving systems MetaPRL, NuPRL and PVS, with intent to include material from other implemented logics such as Minlog, Coq, HOL, Isabelle, and Larch in due course.

The MetaPRL logics that an FDL user is interested in are specified during the build of MetaPRL. After the FDL is connected to MetaPRL, one can retrieve the modules of those logics, and their contents. The data is transferred over TCP sockets in the MathBus interchange format [53]. FDL sends MetaPRL commands that specify what to import and how, and can contain additional evaluation requests. Example commands include listing all modules, retrieving a particular proof in a module, calling the proof engine on a particular proof step, or migrating an entire module, or logic.

For the purpose of the FDL, we typically migrate all the available data. Then, the FDL can check the proofs by calling the MetaPRL proof engine and build the appropriate certificates.

7 Related Work

In parallel with MetaPRL, Cornell PRL group also developed another descendant of NuPRL-4 — NuPRL LPE [5]. These two projects are intended to compliment each other. In particular, NuPRL LPE features a complex implementation of a knowledge base that allows one to store logical objects with arbitrary relations between them — such as, for example, the MetaPRL objects organized in a hierarchy of theories (see Section 5.1). NuPRL LPE's distributed nature allows one to use different logical engines from NuPRL LPE — including the fast logical engine (see Section 3.1) provided by MetaPRL. NuPRL LPE also provides a complex GUI — a logical navigator, which compensates for the lack of an advanced GUI in MetaPRL. NuPRL LPE is currently being used in UAV system protocol verification and in work on practical reflection [9].

MetaPRL has much in common with the Isabelle generic theorem prover [45, 46]. The main differences are the logical foundations and the theory mechanism. We have kept a Martin-Löf style logic, hence the need for computational rewrites. Also, our module mechanism stresses relations between theories, allowing reuse of proof automation.

Harrison's HOL-Light [22] shares some common features with the MetaPRL implementation. Harrison's system is implemented in Caml-Light, and both systems require fewer computational resources than their predecessors.

For a more detailed overview of the work related to some of the individual features of the MetaPRL system, please see the corresponding papers cited above [7,21,24,27,28,29,31,34,35,42,43,44,48,51,52].

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2. Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definition. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986.
3. Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical Report 40, Mittag-Leffler, 2000/2001.
4. Stuart Allen, Mark Bickford, Robert Constable, et al. FDL: A prototype formal digital library. PostScript document on website, May 2002. `http://www.nuprl.org/html/FDLProject/02cucs-fdl.html`.
5. Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The NuPRL open logical environment. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.
6. David Aspinall. Proof General – A generic tool for proof development. In *Proceedings of TACAS*, volume 1785 of *Lecture Notes in Computer Science*, 2000. `http://zermelo.dcs.ed.ac.uk/home/da/papers/pgoutline/`.
7. Brian Aydemir, Adam Granicz, and Jason Hickey. Formal design environments. In Carreño et al. [16], pages 12–22.
8. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual.* INRIA-Rocquencourt, CNRS and ENS Lyon, 1996.
9. Eli Barzilay and Stuart Allen. Reflecting higher-order abstract syntax in NuPRL. In Carreño et al. [16], pages 23–32.
10. J. L. Bates. *A Logic for Correct Program Development.* PhD thesis, Cornell University, 1979.
11. J. L. Bates and Robert L. Constable. Definition of micro-PRL. Technical Report 82–492, Cornell University, Computer Science Department, Ithaca, NY, 1981.
12. J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.
13. W. Bibel. *Automated Theorem Proving.* Vieweg Verlag, Braunschweig, 2nd edition, 1987.
14. Mark Bickford, Christoph Kreitz, Robbert van Renesse, and Xiaoming Liu. Proving hybrid protocols correct. In Richard Boulton and Paul Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 105–120, Edinburgh, Scotland, September 2001. Springer-Verlag.
15. Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors. *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science.* Springer-Verlag, 2002.
16. Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors. *Theorem Proving in Higher Order Logics; Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002), Hampton, VA, August 2002.* National Aeronautics and Space Administration, 2002.
17. Robert L. Constable. On the theory of programming logics. In *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing,* Boulder, CO., pages 269–85, May 1977.

18. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Development System.* Prentice-Hall, NJ, 1986.
19. Robert L. Constable and Jason Hickey. NuPRL's class theory and its applications. In Friedrich L. Bauer and Ralf Steinbrueggen, editors, *Foundations of Secure Computation*, NATO ASI Series, Series F: Computer & System Sciences, pages 91–116. IOS Press, 2000.
20. Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, NY, 1979.
21. Adam Granicz and Jason Hickey. Phobos: A front-end approach to extensible compilers. In *36th Hawaii International Conference on System Sciences.* IEEE, 2002.
22. John Harrison. HOL Light: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.
23. Mark Hayden. *The Ensemble System.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, January 1998.
24. Jason Hickey, Aleksey Nogin, Adam Granicz, and Brian Aydemir. Formal compiler implementation in a logical framework. Submitted to ICFP'03. Extended version of the paper is available as Caltech Technical Report caltechCSTR:2003.002, 2003.
25. Jason Hickey, Justin D. Smith, Brian Aydemir, Nathaniel Gray, Adam Granicz, and Cristian Tapus. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR 2002.007, California Institute of Technology, Computer Science, July 2002.
26. Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the `http://www.cis.upenn.edu/ bcpierce/FOOL/FOOL3.html`.
27. Jason J. Hickey. NuPRL-Light: An implementation framework for higer-order logics. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 395–399. Springer, July 13–17 1997. An extended version of the paper can be found at `http://www.cs.caltech.edu/ jyh/papers/cade14_nl/default.html`.
28. Jason J. Hickey. Fault-tolerant distributed theorem proving. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 227–231, Berlin, July 7–10 1999. Trento, Italy.
29. Jason J. Hickey. *The MetaPRL Logical Programming Environment.* PhD thesis, Cornell University, Ithaca, NY, January 2001.
30. Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`.
31. Jason J. Hickey and Aleksey Nogin. Fast tactic-based theorem proving. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2000.
32. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.
33. Paul B. Jackson. *Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, Ithaca, NY, January 1995.

34. Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science*, 2003. To appear.
35. Alexei Kopylov and Aleksey Nogin. Markov's principle for propositional type theory. In L. Fribourg, editor, *Computer Science Logic, Proceedings of the 10th Annual Conference of the EACSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 570–584. Springer-Verlag, 2001.
36. Christoph Kreitz and Jens Otten. Connection-based theorem proving in classical and non-classical logics. *Journal for Universal Computer Science, Special Issue on Integration of Deductive Systems*, 5(3):88–112, 1999.
37. Christoph Kreitz and Stephan Schmitt. A uniform procedure for converting matrix proofs into sequent-style systems. *Journal of Information and Computation*, 162(1–2):226–254, 2000.
38. Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 34 of *Operating Systems Review*, pages 80–92, December 1999.
39. Conal L. Mannion and Stuart F. Allen. A notation for computer aided mathematics. Department of Computer Science TR94-1465, Cornell University, Ithaca, NY, November 1994.
40. A.A. Markov. On constructive mathematics. *Trudy Matematicheskogo Instituta imeni V.A. Steklova*, 67:8–14, 1962. In Russian. English Translation: A.M.S. Translations, series 2, vol.98, pp. 1-9. MR 27#3528.
41. Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
42. Aleksey Nogin. Quotient types: A modular approach. In Carreño et al. [15], pages 263–280. Available at `http://nogin.org/papers/quotients.html`.
43. Aleksey Nogin. *Theory and Implementation of an Efficient Tactic-Based Logical Framework.* PhD thesis, Cornell University, Ithaca, NY, August 2002.
44. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Carreño et al. [15], pages 281–297.
45. L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical report, University of Cambridge Computing Laboratory, 1990.
46. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1994.
47. Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2. Elsevier Science Publishers, 2001.
48. Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Aleksey Nogin. JProver: Integrating connection-based theorem proving into interactive proof assistants. In *International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 421–426. Springer-Verlag, 2001.
49. The Nuprl Staff. PRL: Proof refinement logic programmer's manual (Lambda PRL, VAX version). Cornell University, Department of Computer Science, 1983.
50. Pierre Weis and Xavier Leroy. *Le langage Caml.* Dunod, Paris, 2nd edition, 1999. In French.
51. Xin Yu. Formalizing abstract algebra in constructive set theory. Master's thesis, California Institute of Technology, 2002.

52. Xin Yu and Jason J. Hickey. Formalizing abstract algebra in constructive set theory. Submitted to LICS conference, 2003.
53. Richard Zippel. MathBus. Available online at `http://www.cs.cornell.edu/Simlab/papers/mathbus/mathTerm.htm`.