System Description
MetaPRL — A Modular Logical Environment *

Aleksey Nogin

Department of Computer Science
Cornell University
Ithaca, NY 14853
nogin@cs.cornell.edu

Jason Hickey

Department of Computer Science
Caltech 256-80
Pasadena, CA 91125
jyh@cs.caltech.edu

1 Introduction

NuPRL 4 [2] is a mature, higher-order theorem prover that has been used extensively, most recently for the verification and optimization of the Ensemble group-membership system [8, 10]. NuPRL is also a pretty big system (about 100,000 lines of Lisp and ML code, some of which is more than a decade old) and, as a result, making substantial changes to NuPRL system, especially the design, can be really hard.

To address this problem we developed a new system, called MetaPRL. MetaPRL is a *tactic* prover, and it is written in OCaml [9]. MetaPRL keeps many of the major design principles and concepts of NuPRL system. For example the two systems have very similar term syntax and MetaPRL implements the NuPRL type theory as one of its logics (see section 3).

However MetaPRL is substantially different from NuPRL 4 and has many new features. MetaPRL (which used to be called NuPRL-Light) is also described in [6]. In this paper we present an overview of the features that were introduced in MetaPRL and that did not exist in previous generations of PRL systems.

MetaPRL documentation and download instructions can be found at
`http://jump.to/meta-prl`

2 Fast Logical Engine

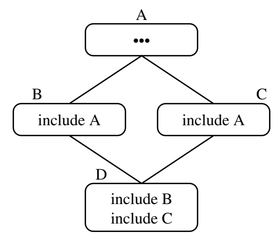
MetaPRL was designed with efficiency in mind. In addition, MetaPRL code is highly modular, which has made it easy to improve the efficiency of the procedures along the critical path (the rewriting engine). MetaPRL modularity has also allowed us to replace generic modules with domain-specific implementations that improve performance in some logics. See [4] for the description of the techniques that were used for speeding up MetaPRL.

As a result, the MetaPRL logical engine is considerably faster than NuPRL 4. We compared two systems by writing tactics that implement some simple domain-specific proof search algorithm in each of the systems. We performed

* This work was partially supported by DARPA grant F30602-98-2-0198

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 (see Section 6) we were able to achieve even greater speed-ups.

3 Hierarchical Theories Mechanism



MetaPRL does not assume any particular type theory or logic but allows users to formulate and use different logics and theories. In MetaPRL mathematical theories are implemented as *theories*, which are extensions of ML modules. Each theory is usually a sequence of logical objects (such as rules, abstraction definitions, etc.) and ML code (such as tactics implementations and tactics improvements). 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 4) that are valid in a superclass are valid in a subclass. In the diagram above we describe a scenario where theories B and C both include theory A and theory D includes both B and C. Only one copy of A will be provided in D with theories B and C sharing a common implementation.

Such a modular mechanism has many advantages. It allows users to formulate 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 a product types and some extra objects, she can take the product type theory from the NuPRL-style type theory implemented in MetaPRL and add all the necessary theories. The 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 that work with product types.

See [6] for a detailed description of MetaPRL logical framework.

4 Derived Rules

NuPRL 4 allowed a user to prove a formula and to instantiate it as a lemma in later proofs. For instance, a proof of a formula $A \Rightarrow B$ could later be used during backward-chaining to reduce a goal sequent $\Gamma \vdash B$ to the subgoal $\Gamma \vdash A$ *and an additional subgoal that* A *is well-formed:* $\Gamma \vdash A$ *Prop*.

MetaPRL extends this mechanism to derived rules. Let us explain why such mechanism is useful.

At first glance it may seem possible to model a derived rules mechanism in NuPRL by replacing meta-implications with logical implications — for example, using a theorem $A \Rightarrow B$ instead of a derived rule "if A is derived, B may be derived". However this is not always possible since the rule and the theorem may have different logical meanings. For example in NuPRL type theory, such a rule would mean, in particular, that well-formedness of A implies well-formedness of

B while the theorem would mean that both A and B are well-formed. Also, the theorem says that A always implies B while the rule states that A only has to imply B when A is provable.

One could also attempt to model derived rules mechanism by writing a tactic that would apply the primitive rules or tactics in the same order as they are applied in the proof of a derived rule. Such approach would work, but would be very inefficient since in order to use such a "derived rule", one would have to run the tactic which would make all the "small steps" instead of making a single "large step", as it is implemented in MetaPRL.

Together with the modular theories mechanism, the derived rules enable *logical abstraction*. For example, the standard definitions of product and function spaces can be derived from the "very"-dependent function space [5]. In MetaPRL, the theories for the simple types are defined as if they were primitive, and then they are derived from the very-dependent theory. The derivation is transparent; simple types may be used as if they were primitive ones. Since MetaPRL also allows rules to be used before they are proved, a user can state rules she knows to be true, use them, and later justify them (or ask somebody else to do it). For accounting purposes, the MetaPRL proof manager classifies rules as primitive (for axioms), unproved, and proved.

5 Computational Rewrites

In MetaPRL it is possible to define not only logical rules, but also logical rewrites. A logical rewrite states that an equivalence between terms that 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 — the rewrites that state that two terms can be interchanged in some context when a certain condition is true in that context. For example, $x \neq 0 \longrightarrow x/x \longleftrightarrow 1$.

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

6 Transparent Concurrent and Distributed Refinement

MetaPRL is capable of distributing a proof search over several processors using the Ensemble group communication system [3]. The distribution it transparent for both the tactic programmer and the system user. That is, the tactics are programmed using the same language as in NuPRL 4 without restriction. Processes may join and leave (even fail) at any time, affecting only changes in speed. Speed improvements are usually linear in the number of processors participating in a proof.

The distribution mechanism is described in-depth in [7]

7 Resources

Tactics implement automated-proving procedures and heuristics. Even with a fixed base logic, as in NuPRL 4, 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 were defined or proved.

MetaPRL automates this process through a mechanism called *resources*. A resource implements an *inheritance* mechanism based on the logical hierarchy (see Section 3) For instance, MetaPRL manages resources for rules, computational rewrites, and special-purpose proof automation. The resource for propositional logic contains the rules for propositional reasoning (like resolution). Since predicate-logic is an *extension* of propositional logic, all the propositional rules are inherited, additional rules are added for the quantifiers, and the resulting resolution procedure is updated. MetaPRL enforces and automates management of the hierarchy, including resources, so that the predicate rules are valid only in the predicate logic and its extensions.

Each resource can be viewed abstractly as a collection. Each resource defines 1) an *insertion* operation for adding an object to the collection (in MetaPRL this is called *improving* the resource), 2) a *join* operation for composing collections when theories are inherited, and 3) an *access* operation for retrieving the values in the collection. Resources are used most commonly to collect the rules and reasoning steps that are later combined into tactics for automated reasoning.

8 Resource Annotations

It turned out that most resource improvements are rather uniform, and it is often possible to automate resource insertions when rules are rewrites are defined. In MetaPRL, automatic procedures are implemented for several resources based on source annotations. The MetaPRL compiler processes the annotations and automatically generates code for resource modifications.

Derived rules and resource annotations together give a new way of implementing many complex tactics. Instead of writing large pieces of code which may be hard to debug and to understand, MetaPRL user can view a tactic as a number of deterministic sequences of rule applications and some control information that specifies which sequences get executed and in which order. And the sequences would be implemented by using the derived rules while the control information would be added as resource annotations on those rules.

9 Related Work

In parallel with MetaPRL, PRL group also developed another descendant of NuPRL 4 — NuPRL 5 [1]. These two projects are intended to compliment each other. In particular, NuPRL 5 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 3). NuPRL 5 distributed nature allows one to use different logical engines from NuPRL 5 — including the fast logical engine (see section 2) provided by MetaPRL. NuPRL 5 also provides a complex GUI — a logical navigator, which compensates the lack of any advanced GUI in MetaPRL.

MetaPRL has much in common with the Isabelle generic theorem prover [11, 12], 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 re-use of proof automation.

References

1. S. F. Allen, R. L. Constable, R. Eaton, C. Kreitz, and L. Lolorigo. Nuprl 5 — an open logical environment. Submitted to CADE-17, 2000.
2. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.
3. Mark Hayden. *The Ensemble System*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1998.
4. Jason Hickey and Aleksey Nogin. Fast tactic-based theorem proving. Unpublished, see `http://www.cs.caltech.edu/~jyh/papers/cade16_fast/default.html`.
5. Jason J. Hickey. Objects and theories as very dependent types. In *Proceedings of FOOL 3*, July 1996.
6. 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 on Artificial Intelligence, pages 395–399, Berlin, July 13–17 1997. Springer. CADE '97. An extended version of the paper can be found at `http://www.cs.caltech.edu/~jyh/papers/cade14_nl/default.html`.
7. 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. Springer. CADE '99.
8. Christoph Kreitz, Mark Hayden, and Jason Hickey. A proof environment for the development of group communications systems. In *Fifteen International Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, pages 317–332. Springer, 1998.
9. Xavier Leroy. *The Objective Caml system release 1.07*. INRIA, France, May 1997.
10. Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17^{th} ACM Symposium on Operating Systems Principles*, December 1999.
11. L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical report, University of Cambridge Computing Laboratory, 1990.
12. L. Paulson and T. Nipkow. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, Vol. 825, 1994.