Fast Tactic-based Theorem Proving *

Jason Hickey

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

Aleksey Nogin

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

Abstract. Theorem provers for higher-order logics often use *tactics* to implement automated proof search. Tactics use a general-purpose meta-language to implement both general-purpose reasoning and computationally intensive domain-specific proof procedures. The generality of tactic provers has a performance penalty; the speed of proof search lags far behind special-purpose provers. We present a new modular proving architecture that significantly increases the speed of the core logic engine. Our speedup is due to efficient data structures and *modularity*, which allows parts of the prover to be customized on a domain-specific basis. Our architecture is used in the MetaPRL logical framework, with speedups of more than two orders of magnitude over traditional tactic-based proof search.

1 Introduction

Several provers [8,9,3,11,12,15,18] use higher-order logics for reasoning because the expressivity of the logics permits concise problem descriptions, and because meta-principles that characterize entire classes of problems can be proved and re-used on multiple problem instances. In these provers, proof automation is coded in a *meta-language* (often a variant of ML) as *tactics*. Automation speed has a direct impact on the level of reasoning. If proof search is slow, more interactive guidance is needed to prune the search space, leading to excessive detail in the tactic proofs.

We present a proving architecture that addresses the problem of speed and customization in tactic provers. We have implemented this architecture in the MetaPRL logical framework, achieving more than two orders of magnitude speed-up over the existing NuPRL-4 implementation. We obtain the speedup in two parts: our architecture is modular, allowing components to be replaced with domain-specific implementations, and we use efficient data structures to implement the proving modules.

In this paper, we explain this modular architecture. We show that the logic engine can be broken into three modules: a *term* module that implements the logical *language*, a term *rewriter* that applies primitive inferences, and a *proof* module that manages proofs and defines tactics. The computational behavior of

* Support for this research was provided by DARPA grants F30602-95-1-0047 and F30602-98-2-0198

proof search is dominated by term rewriting and operations on terms, and we present implementations of the modules for domains with frequent applications of substitution (like type theory), and for domains with frequent applications of unification (like first-order logic).

MetaPRL, our testbed, is implemented in Objective Caml [19]. It includes logics like first-order logic, the NuPRL type theory, and Aczel's CZF set theory [1]. We include performance measurements that compare MetaPRL's performance with NuPRL-4 on the NuPRL type theory. In our measurements, we also show how particular module implementations change the performance in the different domains.

One might think that the comparison between MetaPRL and NuPRL-4 is not very fair since NuPRL-4 uses interpreted ML and MetaPRL is implemented in OCaml. But in fact only very high-level code uses interpreted ML in NuPRL-4 while most of the time is spent performing low-level operations such as term operations and primitive rule applications. And in NuPRL-4 all the low-level operations are implemented in Lisp and are compiled by a modern Lisp compiler. This should make the comparisons relatively fair, especially when we are talking about two orders of magnitude speed difference.

It should also be noted that MetaPRL is a *distributed* prover [14], leading to additional speedups if multiple processors are used. Distribution is implemented by inserting a scheduling and communication layer between the refiner and the tactic interface. For this paper, we describe operation and performance without this additional scheduling layer.

The organization of the paper is a follows. In Section 2, we give an overview of tactic proving, and present the high-level architecture. In Sections 3, 4, and 5, we explore the proving modules in more detail, and develop their implementations. In Section 6, we compare the performance of the different implementations, and in Section 7 we summarize our results, and present the remaining issues. This work builds on the efforts of many systems over the last decade, and in Section 8 we present related work.

2 Architectural Overview

We consider a general architecture of a tactic prover consisting of three parts, as shown in Figure 1. A *logic* contains the following kinds of objects:

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

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

3. *Rewrites* define computational equivalences. For example, the type theory defines functions and application, with the equivalence $(\lambda x.b)\ a \longleftrightarrow b[a/x]$.
4. *Theorems* provide proofs for derived inference rules and axioms.

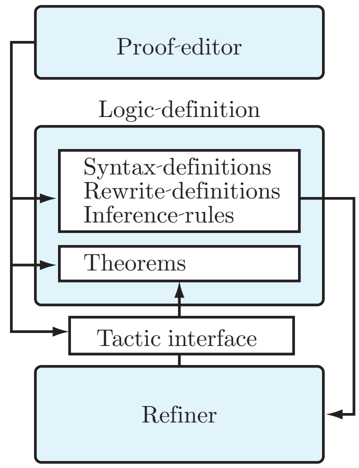


Fig. 1. General tactic prover architecture

The *refiner* [5] performs two basic operations. First, it builds automation procedures from the parts of a logic.

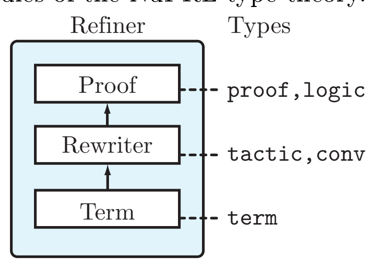
1. Syntax definitions are compiled to functions for constructing logical formulas.
2. Rewrite primitives (and derived rewrite theorems) are compiled to *conversions* that allow computational reductions to be applied during a proof.
3. Inference rules and theorems are compiled to primitive *tactics* for applying the rule, or instantiating the theorem.

The second refiner operation is the *application* of conversions and tactics, producing justifications from the proofs. The major parts of the refiner interface are shown below.[1] It defines abstract types for data structures that implement terms, tactic and rewrite definitions, proofs, and logics. Proof search is performed in a backward-chaining goal-directed style. The `refine` function takes a `logic` and a `tactic` search procedure, and applies it to a *goal* term to produce a *partial* `proof`. The goal and the resulting *subgoals* can be recovered with the `sub/goal_of_proof` projection functions. Proofs can be composed with the `compose proof subproofs` function, which requires that the goals of the `subproofs` correspond to the subgoals of the `proof`, and that both derivations occurred in the same logic. If an error occurs in any of the refiner functions, the `RefineError` exception is raised. The `tactic_of_conv` function creates a tactic from a rewrite definition. The final two functions, called *tacticals*, are the primitives for implementing proof search. Operationally, the `andthen tac1 tac2` tactic applies `tac1` to a goal and immediately applies `tac2` to all the subgoals, composing the result. The `orelse tac1 tac2` is equal to `tac1` on goals where `tac1` does not produce an error, otherwise it is equivalent to `tac2`.

[1] Throughout this paper we will use a simplified OCaml syntax to give the component descriptions.

```
module type RefinerSig = sig
    type term, tactic, conv, proof, logic
    exception RefineError
    val refine : logic → tactic → term → proof
    val goal_of_proof : proof → term
    val subgoals_of_proof : proof → term list
    val compose : proof → proof list → proof
    val tactic_of_conv : conv → tactic
    val andthen : tactic → tactic → tactic
    val orelse : tactic → tactic → tactic
end
```

The `logic` data type is the concrete representation of a logic. The MetaPRL logical framework defines multiple logics in an inheritance hierarchy (partial order) where if L_{child}:`logic` inherits from L_{parent}:`logic`, all the theorems of L_{parent} are valid (and provable) in L_{child}. In contrast, the NuPRL-4 prover has a single global logic containing the syntax and rules of the NuPRL type theory.



In a prover like NuPRL-4, the refiner can be characterized as *monolithic*. There is no well-defined separation of the refiner into components, and there is no well-defined interface like the `RefinerSig` we defined above—there is *one* built-in refiner. This has made it difficult to customize and maintain NuPRL-4, and our choice in MetaPRL has been to partition the refiner into several small well-defined parts.

This modular structure has an additional benefit: if we partition the refiner into abstract parts, we can create domain-specific implementations of its parts. While the whole refiner is a part of a trusted code base, we do not need to worry about introducing bugs while doing domain-specific optimization. When we need to be extra sure, that everything is correct, we can do proof development using the domain-specific code and later double-check the proof using the reference implementation. And for some parts of the system we even have a debugging mode that runs two implementations side-by-side and notifies the user if they behave differently. This not only protects us from bugs introduced by the domain-specific code, but also helps us to debug the reference implementation as well.

The choice of partitioning we use is guided by the *type* definitions, producing the layered architecture shown at the right. The lowest layer, the *term* module, defines the logical language; the *rewriter* module implements applications of primitive tactics and conversions using term rewriting; and the *proof* module defines the `logic` and `proof` data types. We present specifications and implementations of these modules in the following section.

3 The *term* module

All logical terms, including goals and subgoals, are expressed in the language of *terms*, implemented by the *term* module. The general syntax of all terms has three parts. Each term has 1) an operator-name (like "sum"), which is a unique name indicating the logic and component of a term; 2) a list of parameters representing constant values; and 3) a set of subterms with possible variable bindings. We use the following syntax to describe terms, based on the NuPRL definition [2]:

$$\underbrace{\mathit{opname}}_{\mathit{operator\ name}} \quad \underbrace{[p_1; \cdots ; p_n]}_{\mathit{parameters}} \underbrace{\{\boldsymbol{v_1}.t_1; \cdots ; \boldsymbol{v_m}.t_m\}}_{\mathit{subterms}}$$

A few examples are shown at the right. Variables are terms with a string parameter for their name; numbers have an integer parameter. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b.

Displayed form	Term
1	`number[1]{}`
$\lambda x.b$	`lambda[]{x. b}`
$f(a)$	`apply[]{f; a}`
v	`variable["v"]{}`
$x + y$	`sum[]{x; y}`

The term module implements several basic term operations: substitution ($b[a/x]$) of a term (a) for a variable (x) in a term (b), free-variable calculations, α-equivalence, etc. When a logic defines a rule, the refiner compiles the rule pattern into a sequence of term operations. The term interface is shown below. The abstract types `opname`, `param`, `term`, and `bound_term` represent operator names, constant parameters, terms, and bound terms (the subterms of a term). The major operations include destructors to decompose terms and bound-terms, as well as a substitution function `subst`, free variable calculations, and term equivalence.

```
module type TermSig = sig
    (* Types and constructors: *)
    type opname, param, term, bound_term
    val mk_opname : string list → opname
    val mk_int_param : int → param
    val mk_string_param : string → param
    val mk_term : opname → param list → bound_term list → term
    val mk_bterm : string list → term → bound_term

    (* Destructors and other operations: *)
    val dest_term : term → opname * param list * bound_term list
    val dest_bterm : bound_term → string list * term
    val subst : (string * term) list → term → term
    val free_vars : term → string list
    val alpha_equal : term → term → bool
end
```

3.1 Naive term implementation (`Term_std`)

The most immediate implementation of terms is the naive "standard" implementation, which builds the term with tupling.

```
type opname = string list
and param = Int of int | String of string
and term = opname * param list * bound_term list
and bound_term = string list * term
```

While this structure is easy to implement, it suffers from poor substitution performance. The following pseudo-code gives an outline of the substitution algorithm.

```
let rec subst sub t =
    if t is a variable then
        if (t, t') ∈ sub then t' else t
    else let (opname, params, bterms) = t in
        (opname, params, List.map (subst_bterm sub) bterms)
and subst_bterm sub (vars, t) =
    let sub' = remove (v, t') from sub if v ∈ vars in                  ①
    let vars', sub'' = rename binding variables to avoid capture in    ②
    (vars', subst sub'' t)
```

The `sub` argument is a list of `string/term` pairs that are to be simultaneously substituted into the term in the second argument. The main part of the substitution algorithm is in the part for substituting into bound terms. In step ①, the substitution is modified by removing any `string/term` pairs that are freshly bound by the binding list `vars`, and in step ②, the binding variables are renamed if they intersect with any of the free variables in the terms being substituted.

Roughly analyzed, this algorithm takes time at least linear in the size of the term on which the substitution is performed. Furthermore, each substitution performs a full copying of the term. Substitution is a very common operation in MetaPRL — each application of an inference rule involves at least one substitution.[2] The next implementation performs lazy substitution, useful in domains like type theory.

3.2 Delayed substitution (`Term_ds`)

If substitution is frequent, it is often more efficient to save computations for use in multiple substitution operations. We use three main optimizations: we save free-variable calculations, we perform lazy substitution, and we provide special representations for commonly occurring terms.

When a substitution is performed on a term for the first time, we compute the set of free variables of that term, and save them for later use. When a substitution is applied, the free-variables set is used to discard the parts of the substitution for variables that do not occur free in the term. This saves time, and it also saves space by reusing subterms where the substitution has no effect

[2] Testing for α-equivalence also takes linear time. One way to decrease the cost would be to use a normalized representation (such as a DeBruijn representation). However, term *destruction* on the normalized representation can be expensive because of the need to rename variables that become free (what are the subterms of $\lambda x.\lambda y.xy$ and $\lambda x.\lambda y.yx$?). These renamings can be delayed, as the next Section shows, but the cost of equivalence testing will increase.

instead of unnecessarily copying them. Memory savings, in turn, further improve performance by improving the CPU cache efficiency and reducing the GC time.

During proof search, most tactic applications fail, and only a part of the substitution result is usually examined in the proof search. In this common case, it is more efficient to delay the application of a substitution until the substitution results are actually requested by the `dest_term` function.

We also optimize two commonly-occurring terms: *variables* and *sequents*. Rather than using the term encoding of variables, we provide a custom representation using a string. The *sequent* optimization uses a custom data structure to give constant-time access to the hypotheses, instead of the usual linear-time encoding. These "custom" terms are abstract optimizations—they do not change the `Term` interface definition. For each custom term, we add special-case handlers to each of the generic term functions.

The following definition of terms uses all of these optimizations (the definitions for the `bound_term`, `opname` and `param` types are unchanged). The definition of sequents, which we omit, uses arrays to represent the hypotheses and conclusions of the sequent.

```
type term = { free_vars :   VarsDelayed
                          | Vars of string set;
              core :   Term of (opname * param list * bound_term list)
                     | Subst of (subst * term)
                     | Var of string
                     | Sequent of sequent }
and subst = (string * term) list
and sequent = ···
```

The `free_vars` field caches the free variables of the term, using `VarsDelayed` as a placeholder until the variable set is computed. The `core` field stores the term value, using the `Term` variant to represent values where a substitution has been expanded, the `Subst` variant to represent delayed substitutions, and the `Var` and `Sequent` variants for custom terms. We maintain the following invariants on `Subst`: substitution lists are never empty, and the domain of the substitution is included in the free-variables of the term.

The free-variables computation is one of the more complex operations on this data structure. When the free variables are computed for a term, there are three main cases: if the free variables have already been computed, they are returned; if the `core` is a `Term`, the free variables are computed from the subterms; and if the `core` is a delayed substitution, the substitution is used to modify the free variables of the inner term.

```
let rec free_vars = function
      { free_vars = Vars fv } → fv
    | { core = core } as t →
        let fv = match core with
              Term (_, _, bterms) → Set.map_list free_vars_bterm bterms
            | Subst (sub, t) → free_vars_subst sub (free_vars t)
            | Var v → Set.singleton v
            | Sequent seq → free_vars_sequent seq
        in (t.free_vars ← Vars fv); fv
```

```
and free_vars_bterm (bvars, t) =
    Set.subtract_list (free_vars t) bvars
and free_vars_subst sub fv =
    Set.union
        (Set.subtract_list fv (List.map fst sub))
        (Set.map_list free_vars (List.map snd sub))
```

If the free variables haven't already been computed, the `free_vars` function computes them, and assigns the value to the `free_vars` field of the term. In the `Term` case, the free variables are the union of the free variables of the subterms, where any new binding occurrences have been removed. In the `Subst (sub, t)` case, the free variables are computed for the inner term `t`, then the variables being replaced are removed from the resulting set, and then the free variables of the substituted terms are added.

The `subst` function has a simple implementation: eliminate parts of the substitution that have no effect (in order to maintain the invariant), and save the result in a `Subst` pair if the resulting substitution is not empty.

```
let subst sub t =
    let fv = free_vars t in
        match remove (v, t') from sub if v ∉ fv with ①
          [] → t (* substitution has no effect *)
        | sub' → { free_vars = VarsDelayed; core = Subst (sub', t) }
```

The `set` implementation determines the complexity of substitution. If the `set` lookup takes $O(1)$, then pruning ① takes time linear in the number of variables in `sub`.

The effect of the substitution is delayed until the term is destructed. The `dest_term` function is required to expand the substitution by one step. We use the `get_core` function, shown below, to expand the toplevel substitutions in the term. If the substitution was applied to a `Term`, `get_core` will push it down to the immediate subterms. After the substitution is expanded, `get_core` will store the result in the `core` field to save time on the next `get_core` invocation. As usual, we omit the code for sequents.

```
let rec get_core = function
  { core = Subst (sub, t') } as t →
    let core' = match get_core t' with
          Var v → get_core (List.assoc v sub) (* always succeeds *)
        | Term (opname, params, bterms) →
             Term (opname, params, List.map (do_bterm_subst sub) bterms)
        | Sequent seq → Sequent (sequent_subst sub seq)
    in (t.core ← core'); core'
| { core = simple_core } → simple_core
and do_bterm_subst sub (vars, t) =
    let sub' = remove (v, t) from sub if v ∈ vars in
    let vars', sub'' = rename binding variables to avoid capture in
        (vars', subst sub'' t)
```

Note that the `List.assoc` in the `Var` case will never fail, due to our invariants.

The `dest_term` function first uses `get_core` to expand the top-level substitution (if any), and then it returns the parts of the term. To preserve the external interface of the term module, it is also required to convert the custom terms back to their original form.

```
let rec dest_term t = match get_core t with
      Term (opname, params, bterms) → (opname, params, bterms)
    | Var v → (mk_opname ["variable"], [String v], [])
    | Sequent s → dest_sequent s
```

4 The *rewriter* module

The rewriter performs term manipulations for rule applications. Inference rules and computational rewrites are both expressed using second-order patterns. For example, the rewrite for beta-reduction is expressed with the following pattern:

$$(\lambda x.b_x)\ a \to b_a$$

In this rewrite, the variable a is a *pattern* variable, representing the "argument" term. The variable b_x is a *second-order pattern* variable, representing a term with a free variable x. The pattern b_a represents a *substitution*, with a substituted for x in b. The $(\lambda x.b_x)\ a$ is called the *redex*, and the substitution b_a is called the *contractum*.

```
module Rewrite (Term : TermSig) : sig
    type redex_prog, con_prog, state
    exception RewriteError
    val compile_redex : term → redex_prog
    val apply_redex : redex_prog → term → state
    val compile_contractum : redex_prog → term → con_prog
    val build_contractum : con_prog → state → term
end
```

In NuPRL-4 the computation and inference engines are implemented as separate interpreters that are parameterized by the rewriting patterns. In MetaPRL we combine these functions and improve performance by compiling to a rewriting virtual machine. The MetaPRL rewriter module provides four major functions. The `compile_redex` function takes a redex pattern, expressed as a term, and it compiles it to a redex *program*. The `apply_redex` function applies a precompiled program to a specific term, raising the `RewriteError` exception if the pattern match fails, or returning a `state` that summarizes the result. The `compile_contractum` compiles a contractum pattern against a particular redex program, and the `build_contractum` function takes the contractum program and the result of a redex application, and produces the final contractum term.

Currently, the rewrite module compiles redices to bytecode programs that perform pattern matching, storing the parts of the term being matched in several register files. Contracta are also compiled to bytecode programs that construct the contractum term using the contents of the register file. The virtual machine has the four parts shown in Figure 2:

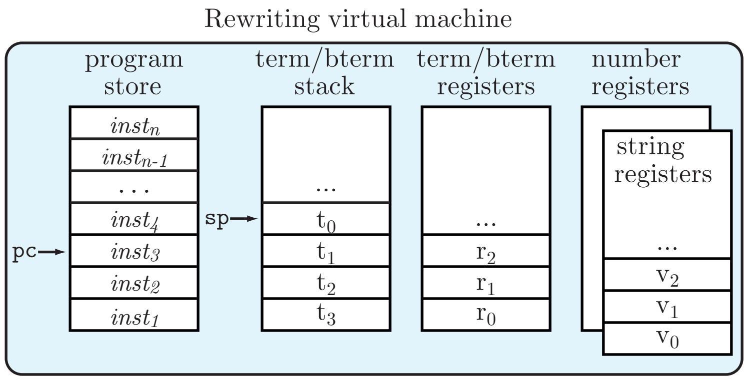


Fig. 2. Rewrite virtual machine

1. a *program* store and program counter for the rewrite program,
2. a *term/bterm stack* with a stack pointer to manage the current term being rewritten,
3. a *term/bterm* register file,
4. a *parameter* register file for each type of parameter.

The instructions for the machine are shown in Figure 3. The matching instruction `dest_term` checks if the term at the top of the term stack has the operator name `opname`, and if it has the right number of bound terms and parameters of the given types. If it succeeds, the parameters are saved in the parameter registers, the term is popped from the term stack, and the bound terms are pushed onto the stack. The `mk_term` instruction does the opposite: it retrieves the parameter values from the register file, pops bc bound terms from the stack, adds the `opname` and pushes the resulting term onto the stack. The `dest_bterm` and `mk_bterm` functions are used to save and restore binding variables for the term at the top of the stack.

The `so_var` instruction pops a term from the term stack and saves it in term register r, along with the free variables in $v_1, \ldots, v_n$. The corresponding constructor `so_subst` pops bc terms from the stack, substitutes them for the variables $v_1, \ldots, v_n$ in term r, and pushes the result onto the stack. The `match_term` instruction is used during matching for redices like $(x + x) \rightarrow 2x$ that contain common subterms.

The example in the Figure gives the code for a beta-reduction. The first `dest_term` instruction matches the outermost `apply` term and pushes the function and argument onto the stack. The `dest_bterm` operations remove the binding variables of the subterms, and the `so_var` instructions stores the results to the register file. At the end of a match against the term $(\lambda x.b)\ a$, register r_1 contains b, register r_2 contains a, and register v_1 contains x. When the contractum is constructed, the first instruction pushes a onto the stack; and the second instruction pops a, substitutes it for x in b, and pushes the result.

Instructions		Example: $(\lambda x.b_x)\ a \longrightarrow b_a$	
Matching:		Redex:	
`dest_term`	`opname`$[p_1;\cdots;p_n].bc$	`dest_term`	`apply`$[].2$
`dest_bterm`	$v_1,\ldots,v_n$	`dest_bterm`	
`match_term`	$r[t_1;\cdots;t_n]$	`dest_term`	`lambda`$[].1$
`so_var`	$r[v_1;\cdots;v_n]$	`dest_bterm`	v_1
Constructors:		`so_var`	$r_1[v_1]$
`mk_term`	`opname`$[p_0;\cdots;p_n].bc$	`dest_bterm`	
`mk_bterm`	$v_1,\ldots,v_n$	`so_var`	$r_2[]$
`so_subst`	$r.bc$		
p_i: parameter register		Contractum:	
v_i: string register		`so_subst`	$r_2.0$
r: term register		`so_subst`	$r_1.1$
t_i: literal term			
bc: arity of bterm			

Fig. 3. Virtual machine instructions

5 The *proof* module

The third part of the refiner manages validity in logics as well as maintaining proof trees for theorems. The proof module exports the interface shown below. The `empty_logic` is the logic without any rules/rewrites. The `join_logics` function builds the union of two logics, and the `add_rule` and `add_rewrite` function add rules/rewrites from their syntactical description as terms. The `proof` type represents a partial proof tree [7], which may be modified by applying a tactic to the proof goal with the `refine` function. The `compose` function is used to stitch together partial proofs into larger proofs. Bookkeeping must be performed here—the proofs being joined must belong to the same logic. If an error occurs in any of the functions, the `RefineError` exception is raised. These functions are not difficult to implement, and we skip the description of their implementations.

```
module Proof (Term : TermSig) (Rewrite : RewriteSig) : sig
    type logic, tactic, rewrite, proof  exception RefineError
    val empty_logic : logic
    val join_logics : logic → logic → logic
    val add_rule : logic → term → logic * tactic
    val add_rewrite : logic → term → logic * rewrite
    val new_proof : term → proof
    val refine : proof → tactic → proof
    val compose : proof → proof list → proof
    val proof_goal : proof → term
    val proof_subgoals : proof → term list
end
```

6 Performance

We group the performance measurements into two parts. All measurements were done on a Linux 400MHz Pentium machine, with 512MB of main memory, and all times are in seconds. For the first part, we compare the speed of the MetaPRL prover (using the modular refiner) with the NuPRL-4 prover. For the first example, we perform pure evaluation based on the following definition of the factorial function:

$$\text{rewrite } \mathit{fact}\{i\} \longleftrightarrow \mathit{if}\ i = 0\ \mathit{then}\ 1\ \mathit{else}\ i * \mathit{fact}\{i-1\}$$

We used the following evaluation algorithm: recursively traverse the term top-down, performing beta-reduction, unfolding the `fact` definition (taking care to evaluate the argument first), etc. This algorithm stresses search during rewriting. Roughly speaking, evaluation should be quadratic in the factorial argument: each term traversal is linear in the size of the term, and the size of the term grows linearly with each traversal (rewriting does not use tail-recursion), until the final base case is reached and the value is computed. The following table lists the performance numbers.

	Argument value			
Configuration	100	250	400	650
`Term_std`	0.35	2.05	5.42	16.0
`Term_ds`	0.42	2.41	6.32	18.4
NuPRL-4	55	330	>1800	>1800

On this example, the NuPRL-4 took between 125 and 160 times longer on the problems where it finished within 30 minutes. On the two larger problems, we terminated the computation after 30 minutes.[3] In MetaPRL, the largest problem performs about 14 million attempted rewrites.

This table also shows a difference between the term module implementations. The "naive" term module performs better on this example because the recursive traversals of the term expand most of the delayed substitutions.

The next example also compares MetaPRL with NuPRL-4, on the pigeonhole problem stated in propositional logic[4]. The pigeonhole problem of size i proves that $i+1$ "pigeons" do not fit into i "holes." The `pigeonT` tactic performs a search customized to this domain, and the `propDecideT` tactic is a generic decision procedure for *intuitionistic* propositional logic (based on Dyckoff's algorithm [10]). Both search algorithms use only propositional reasoning and both explore an exponential number of cases in i.

[3] NuPRL-4 *can* evaluate these terms. The built-in term evaluator, which bypasses the refiner, evaluates the largest example in about 22 seconds.

[4] This formalization of pigeon-hole principle and methods we are using to prove it are obviously highly inefficient. However this formalization provided us with a nice way of comparing the performance of simple propositional proof search in the two systems.

Configuration	Tactic	Problem size 2	3	4	Memory (Max MB)[5]
`Term_std`	`pigeonT`	<0.1	2.53	94.0	126
`Term_ds`	`pigeonT`	<0.1	0.71	17.0	20.8
NuPRL-4	`pigeonT`	0.5	89	>1800	
`Term_std`	`propDecideT`	0.3	238	>1800	
`Term_ds`	`propDecideT`	0.13	55.0	>1800	
NuPRL-4	`propDecideT`	21.9	>1800	>1800	

In this example, NuPRL-4 works between 125 and 170 times slower than `Term_ds`. And the delayed-substitution implementation of terms performs significantly better than the naive implementation, partly because of the efficient substitution in the application of the rules for propositional logic, and also because the `Term_ds` module preserves a great deal of sharing of common subterms. On the largest problem the `pigeonT` tactic performs about 1.57 million primitive inference steps.

For the last examples, we give a few comparisons between the MetaPRL modules in two additional domains. The GEN problems is a heredity problem in a large first-order database. The NUPRL problem is an automated rerun of all proof transcripts in the NuPRL type theory. The transcripts contain a mix of low-level proof steps, such as lemma application and application of inductive reasoning, to higher-level steps that include verification-condition automation and proof search. The transcripts contain about 2,500 interactive proof steps.

Configuration	Problem GEN	NUPRL
`Term_std`	20.4	39.3
`Term_ds`	14.4	36.6

We don't include performance measurements for NuPRL-4 on these examples, because the system differences require a porting effort (for instance, NuPRL-4 does not currently implement a generic first-order proof search procedure). In our experience with NuPRL-4, proofs with several hundred steps tend to take several minutes to replay.

Once again, the `Term_ds` module performs better than the naive terms, due to the frequent use of substitution in applications of the rules of these theories. The times for proof replay include the time spent loading the proof transcripts and building the tactic trees. This cost is similar for both term implementations, and the performance numbers are comparable. The first-order problem, GEN, performs proof search by resolution, using the refiner to construct a primitive proof tree only when a proof is found. This final step is expensive, because each resolution step has to be justified by the refiner. The final successful proof in this problem performs about 41 thousand primitive inference steps.

7 Summary

We have achieved significant speedups for tactic proving. Our new prover design shows consistent speedups of more than two orders of magnitude over the

[5] This does not include the space that the system occupied after the initial loading — 19 MB with `term_std` and 20.5 MB with `term_ds`

NuPRL-4 system. Most of this speedup is due to efficient implementations of the prover components, but an additional part is due to the modular design, which allows the prover to be customized with domain-specific implementations. In addition, the MetaPRL system is programmed in OCaml, an efficient modular language. In contrast, NuPRL-4 tactics are programmed in *classic* ML, which is compiled to Common Lisp, and the NuPRL-4 refiner is implemented in Common Lisp.

In first-order logics, we estimate that an order of magnitude speed factor remains between MetaPRL and provers like ACL2 [17]. Some of this difference can be addressed with a specific refiner modules: a first-order term module would contain custom representations for terms in disjunctive normal form and sequents (sequents provide particularly poor representations for large first-order problems), and the rewrite module would optimize inference by resolution. However, a better solution would be to integrate first-order provers into the logical framework using translation modules that provide a tactic interface through encapsulation of the external functions.

There are a few avenues left to explore. Since we compile rewrites to bytecode, it is natural to wonder what the effect of compiling to native code would be. Also, while we currently do not optimize the proof module, there is significant overhead in composing and saving the primitive proof trees. In some domains, we may be able to perform proof compression, or delay the composition of proofs.

8 Related work

Harrison's HOL-Light [13] 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. Howe [16] has taken another approach to enhancing speed in NuPRL-4. The programming language defined by the NuPRL type theory is *untyped*, leading to frequent production of well-formedness (verification) conditions. Using type annotations, Howe was able to speed up rewriting in NuPRL-4 by a factor of 10. We haven't attempted to apply Howe's ideas to MetaPRL implementation of NuPRL type theory, but we believe that MetaPRL performance can be further improved using these ideas.

Basin and Kaufmann [4] give a comparison between the NuPRL-3 system and NQTHM [6] (the predecessor of the ACL2 [17] system). The NQTHM prover uses a quantifier-free variant of Peano arithmetic. Basin and Kaufmann's measurements showed that NQTHM was roughly 15 times faster than NuPRL-3 for *different formalizations* of Ramsey's theorem. It is likely that ACL2 and NuPRL-4 have a larger gap in speed.

References

1. 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.

2. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 95–197. IEEE, June 1990.
3. 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.
4. David A. Basin and M. Kaufmann. The Boyer-Moore prover and NuPRL: An experimental comparison. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 89–119. Cambridge University Press, 1991.
5. J. L. Bates. *A Logic for Correct Program Development*. PhD thesis, Cornell University, 1979.
6. R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
7. Robert L. Constable, T. Knoblock, and J.L. Bates. Writing programs that construct proofs. *J. Automated Reasoning*, 1(3):285–326, 1984.
8. Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type Theory and Programming. *EATCS*, February 1994. bulletin no 52.
9. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, April 1995. `http://www.csl.sri.com/sri-csl-fm.html`.
10. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. In *The Journal of Symbolic Logic*, volume 57(3), September 1992.
11. R.L. Constable et.al. *Implementing Mathematics in the NuPRL Proof Development System*. Prentice–Hall, 1986.
12. M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
13. John Harrison. HOL Light: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 265–269. Springer LNCS 1166, 1996.
14. Jason Hickey. Fault-tolerant distributed theorem proving. In Harald Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 227–231, Trento, Italy, July 7–10, 1999. Springer-Verlag.
15. Jason J. Hickey. Nuprl-Light: An implementation framework for higher–order logics. In *14th International Conference on Automated Deduction*. Springer, 1997.
16. Douglas J. Howe. A type annotation scheme for Nuprl. In *Theorem Proving in Higher-Order Logics*. Springer, 1998.
17. Matt Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.
18. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer LNCS 828, 1994.
19. Pierre Weis and Xavier Leroy. *Le langage Caml*. Dunod, Paris, 2nd edition, 1999. In French.