Fast Tactic-based Theorem Proving

Jason Hickey and Alexey Nogin

Department of Computer Science, Cornell University

Abstract. Theorem provers for higher-order logics often use *tactics* to code automated proof search. Tactics use a procedural meta-language to perform both algorithmic and heuristic proof search, as well as 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 [7, 8, 10, 16, 11, 13, 6] use higher-order logics for reasoning because the expressivity of the logics can be used to obtain concise models, and because they can express meta-principles that allow results to be re-used on multiple problems. 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 speedup over the existing Nuprl4 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 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).

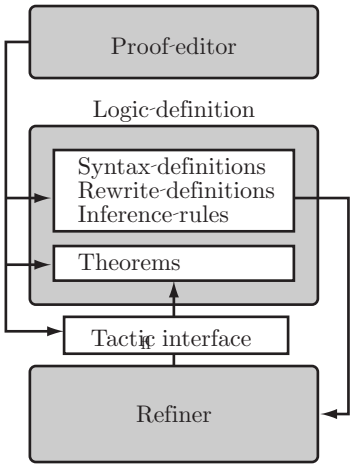


Fig. 1. General tactic prover architecture

MetaPRL, our testbed, is implemented in Objective Caml [17]. 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 Nuprl4 on the Nuprl type theory. In our measurements, we also show how particular module implementations change the performance in the different domains.

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

The general architecture of a tactic prover can be divided into three parts, 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 inference of a logic. For instance, the first-order logic contains rules like AXIOM and MODUS_PONENS in a sequent calculus.

$$\frac{}{\Gamma, a\colon A \vdash A}\ \text{AXIOM} \tag{1}$$

$$\frac{\Gamma, f\colon A \Rightarrow B \vdash A \quad \Gamma, f\colon A \Rightarrow B, b\colon B \vdash C}{\Gamma, f\colon A \Rightarrow B \vdash C}\ \text{MODUS\_PONENS} \tag{2}$$

3. *Rewrites* define computational equivalences. For example, the Nuprl type theory defines λ abstraction and application, with the following equivalence:

$$(\lambda x.b)\ a \longleftrightarrow b[a/x]. \tag{3}$$

4. *Theorems* provide proofs for derived inference rules and axioms.

The *refiner* [4] performs two basic operations. First, from the parts of the logic, it builds the reasoning primitives:

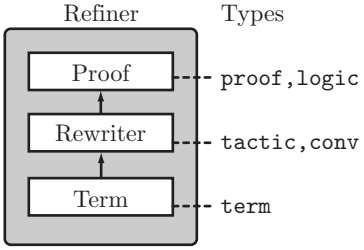
1. Syntax definitions are compiled to sentence constructors and destructors.
2. Rewrite primitives (and derived rewrite theorems) are compiled to *conversions* that allow computational reductions to be applied to terms in a proof.
3. Inference rules and theorems are compiled to primitive *tactics* for applying the rule, or instantiating the theorem.

The second function of the refiner is to implement the *application* of conversions and tactics, producing justifications from the proofs. The major parts of the refiner interface are shown below. 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`.

```
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 *proof editor* provides the user interface. It allows logical components to be viewed and modified, it manages the construction of tactic proofs, and it organizes the presentation of the logic primitives and proofs.

Some provers may not implement all the parts we mention (for instance, the Nuprl4 prover has a "standard" global syntax), but they will in general include a subset of these components. *Logical frameworks* develop this a little farther. In a system like MetaPRL, multiple logics are defined as modules, and the refiner also manages the derivation of one logic from another. However, the definition of the *refiner* (the focus of this paper) is unchanged, except that every a theorem is only allowed to use logical components from the logic in which it was defined.



In a prover like Nuprl4, 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. The first architectural design choice we make is simple: if we partition the refiner into abstract parts, we can create domain-specific implementations of its parts. 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 possibly-bound subterms. We use the following syntax to describe terms, based on the Nuprl definition [2]:

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

Here are a few examples:

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

Variables are terms with a string parameter giving their names; numbers have an integer parameter with their value. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b.

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 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 copy of the term. Substitution is a very common operation—each application of an inference rule involves at least one substitution. 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 in 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 preserving subterms where the substitution has no effect.

During proof search, tactic applications often fail, and only 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.

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 (free_vars t) (Set.of_list bvars)
and free_vars_subst sub fv =
    Set.map (fun v →
        try free_vars (List.assoc v sub) with
        Not_found → Set.singleton v) fv
```

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`, and then the substitution `sub` is applied to resulting set. The `List.assoc sub v` function fetches the substition for `v` from the association list `sub`, raising the exception `Not_found` if there is no entry.

The `subst` function has a simple implementation: eliminate parts of the substitution that have no effect, 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 point that 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.

```
let rec get_core = function
  { core = Subst (sub, t') } as t →
    let core' = match get_core t' with
          Var v → 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 = core } → 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)
```

When the substitution is applied to a `Term`, it is recursively applied to the subterms, performing a single substitution step. Note that the `List.assoc` in the `Var` case will never fail, due to two invariants: substitution lists are never empty, and the domain of the substitution is contained within the free-variables of the term.

The `dest_term` function first uses `get_core` to expand all substitutions, 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 (as usual, we omit the code for sequents).

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

4 The *rewriter* module

The rewriter performs term operations for primitive rule application. It contains a rule compiler that compiles descriptions of inference rules to logical rewriting operations. Inference rules and computational rewrites are expressed using second-order patterns. For example, the rewrite for beta-reduction is expressed with the following pattern:

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

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*.

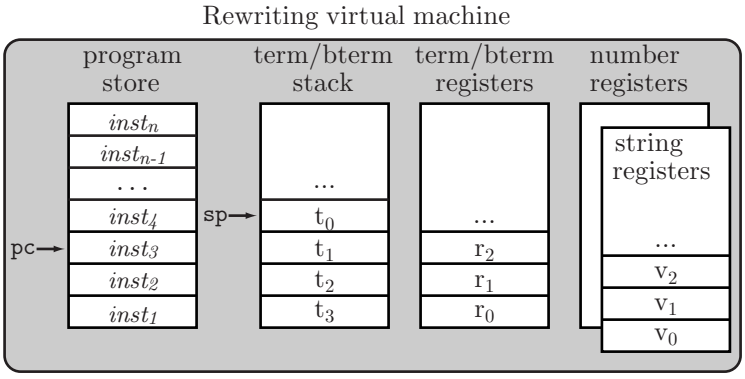


Fig. 2. Rewrite virtual machine

The 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 pre-compiled 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` term takes the contractum program and the result of a redex application, and produces the final contractum term.

```
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
```

Currently, we use a single implementation of the rewrite module, implementing the `Rewrite` interface using a *virtual machine*. Redices are compiled 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:

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

```
Matching:
dest_term  opname[p_1; ···; p_n] bterm_count
dest_bterm v_1, ..., v_n.r
match_term r[t_1; ···; t_n]
so_var     r[v_0; ···; v_n]

Constructors:
mk_term  opname[p_0; ···; p_n] bterm_count
mk_bterm v_1, ..., v_n.r
so_subst r[t_1; ...; t_n]

p_i: parameter register (string or number)
v_1: string register
r: term register
t_i: literal term
```

Fig. 3. Virtual machine instructions

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 bound terms are pushed onto the bterm stack, and the term is popped from the term stack. The `mk_term` instruction does the opposite: it retrieves the parameter values from the register file, pops `bterm_count` bound terms from the bterm stack, adds the `opname` and pushes the resulting term on the term stack. The `dest_bterm` and `mk_bterm` function perform similar operations on bound terms.

The `so_var` instruction copies the term on the term stack to a term register, along with the free variables $v_1, \ldots, v_n$. The corresponding constructor `so_subst` pushes the term in register r, with literal terms $t_1, \ldots, t_n$ substituted for its substitution variables $v_1, \ldots, v_n$ (saved with the term in the term register). The `match_term` instruction is used during matching for redices like $(x + x) \rightarrow 2x$ that contain common subterms.

5 The *proof* module

The third part of the refiner manages 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, 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. Accounting 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 Nuprl4 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	2.25	6.0	17.5
`Term_ds`	0.4	2.7	7.5	21
Nuprl4	55	330	*	*

On this example, the Nuprl4 took between 120 and 150 times longer on the problems where it finished within 30 minutes. On the two larger problems, we terminated the computation after 30 minutes.[1] In MetaPRL, the largest problem performs about 20 million attempted rewrites.

[1] Nuprl4 *can* evaluate these terms. The built-in term evaluator, which bypasses the refiner, evaluates the largest example in about 22 seconds.

This table also shows an interesting 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 Nuprl4, on a the pigeonhole problem stated in propositional logic. The pigeonhole problem of size i proves that $i+1$ "pigeons" do not fit into i "holes." The `pigeonT` tactic performs an efficient search customized to this domain, and the `propDecideT` tactic is a generic decision procedure for *intuitionistic* propositional logic (based on Dyckoff's algorithm [9]). Both search algorithms explore an exponential number of cases in i.

Configuration	Tactic	Problem size 2	3	4	Memory (Max MB)
`Term_std`	`pigeonT`	<0.1	2.46	93.8	170
`Term_ds`	`pigeonT`	<0.1	0.59	15.1	32
`Term_std`	`propDecideT`	0.28	233	>1800	
`Term_ds`	`propDecideT`	0.11	54	>1800	
Nuprl4	`pigeonT`	0.5	89	>1800	
Nuprl4	`propDecideT`	21.9	>1800	>1800	

In this example, 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 1.5 million primitive inference steps.

For the last examples, we give a few comparisons between the MetaPRL modules in three domains. The GEN problems is a heredity problem in a large first-order database. The NUPRL and CZF problems are problems synthesized from the Nuprl type theory, and the CZF set theory. These last two examples are replays of proof transcripts in the higher-order theories. We have combined transcripts from several proofs, and used repetition to expand the proofs to 10 times their original size (the original proof transcripts replayed too quickly to get significant measurements). 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 expanded transcripts correspond to 2,700 interactive steps in the NUPRL example, and 2,300 in CZF.

We don't include performance measurements for Nuprl4 on these examples, because the system differences require a porting effort (for instance, Nuprl4 does not currently implement a generic first-order proof search procedure). In our experience with Nuprl4, proofs with several hundred nodes tend to take several minutes to replay.

Configuration	Problem		
	GEN	NUPRL	CZF
`Term_std`	9.71	26.5	8.08
`Term_ds`	6.75	10.17	5.0

Once again, the `Term_ds` module performs better than the naive terms, due to the frequent use of substitution on the application of the rules of these theories. 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 acheived significant speedups for tactic proving. Our new prover design shows consistent speedups of more than two orders of magnitude over the Nuprl4 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, Nuprl4 tactics are programmed in *classic* ML, which is compiled to Common Lisp, and the Nuprl4 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 [15]. 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 particulary 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 [12] 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 [14] has taken another approach to enhancing speed in Nuprl4. 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 Nuprl4 by a factor of 10.

Basin and Kaufmann [3] give a comparison between the Nuprl3 system and NQTHM [5] (the predecessor of the ACL2 [15] 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 for *different formalizations* fo Ramsey's theorem. It is likely that ACL2 and Nuprl4 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. 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.
4. J. L. Bates. *A Logic for Correct Program Development.* PhD thesis, Cornell University, 1979.
5. R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.
6. Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type Theory and Programming. *EATCS*, February 1994. bulletin no 52.
7. 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`.
8. Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The coq proof assistant user's guide. Technical Report Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.
9. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. In *The Journal of Symbolic Logic*, pages Vol. 57, Number 3, September 1992.
10. R.L. Constable et.al. *Implementing Mathematics in the NuPRL Proof Development System.* Prentice–Hall, 1986.
11. M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.
12. John Harrison. HOL Light: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 265–269. Springer LNCS 1166, 1996.
13. Jason J. Hickey. Nuprl-Light: An implementation framework for higher–order logics. In *14th International Conference on Automated Deduction.* Springer, 1997.
14. Douglas J. Howe. A type annotation scheme for Nuprl. In *Theorem Proving in Higher-Order Logics.* Springer, 1998.
15. 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.

16. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer LNCS 828, 1994.
17. Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.