JFP **14** (1): 3–19, January 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796803004921 Printed in the United Kingdom

Dependent types ensure partial correctness of theorem provers

ANDREW W. APPEL
Princeton University, 35 Olden Street, Princeton, NJ 08544, USA
(*e-mail:* appel@princeton.edu)

AMY P. FELTY
University of Ottawa, 800 King Edward Ave., Ottawa, Ontario K1N 6N5, Canada
(*e-mail:* afelty@site.uottawa.ca)

Abstract

Static type systems in programming languages allow many errors to be detected at compile time that wouldn't be detected until runtime otherwise. Dependent types are more expressive than the type systems in most programming languages, so languages that have them should allow programmers to detect more errors earlier. In this paper, using the Twelf system, we show that dependent types in the logic programming setting can be used to ensure partial correctness of programs which implement theorem provers, and thus avoid runtime errors in proof search and proof construction. We present two examples: a tactic-style interactive theorem prover and a union-find decision procedure.

1 Introduction

Many theorem proving systems implement tactics and tacticals, which provide flexible goal-directed proof search. Tactics reduce goals to subgoals, while tacticals are primitives for combining tactics into larger ones that can perform multiple proof steps. They also allow programming of proof search strategies. Some of the first provers using this style of proof search (e.g. LCF (Gordon *et al.*, 1979) and HOL (Gordon & Melham, 1993)) were written in ML, whose pattern matching, exception handling, and polymorphic type system are useful in writing tactics concisely. Felty (1993) showed that Lambda Prolog's (Nadathur & Miller, 1988) higher-order unification, backtracking, and polymorphic type system provided a more expressive notation for writing tactics and tacticals. Specifically, higher-order abstract syntax is more useful and expressive than ML pattern matching, backtracking is more concise than exception handling, but Lambda Prolog's prenex-polymorphic type system is essentially similar to ML's.

In this paper, we shall discuss the advantages of a dependent type system over ML-style polymorphism for writing theorem provers. Dependent types could be used in a functional language (such as ML) or a logic-programming language (such as Lambda Prolog); we use the logic-programming language Twelf (Pfenning & Schürmann, 1999). This means that the style of prover we illustrate is similar to those presented by Felty (1993), but the issue of ML-style types vs. dependent types is orthogonal to the issue of ML-style or Prolog-style control and data structures. A decade of experience with tactical Felty's prover has shown that this technique is expressive and powerful, and could be used as the core of a full interactive theorem prover similar in strength to many existing provers such as HOL and Isabelle that have been used in a variety of large-scale applications; we expect that the dependently typed variant of Felty's approach would scale just as well.

A problem in the implementation of theorem provers (tactical and other) is that they may have bugs. That is, the "proof" constructed by the prover may not be valid. There are at least two ways that industrial-strength theorem provers defend against invalid proofs:

- Edinburgh LCF (and Isabelle (Paulson, 1994), HOL, etc.) have an unforgeable abstract data type *theorem*. An attempt by a prover to construct an invalid proof will be detected at run time when some (privileged) proof-constructor function detects mismatched arguments.
- Coq (Barras *et al.*, 1998) (and Elf, Twelf, etc.) require provers to construct proof witnesses that can be checked (in principle) by a small and reliable type-checker that's independent of any (complex, unreliable) theorem prover. Provers in Coq and Twelf have usually been written in ML (Caml and Standard ML, respectively); although each of these systems contains a dependently typed language (functional and logic-programming, respectively), that language is meant for describing objects in the object logic, and not as a language for programming provers.

But in each case, bugs in the tactics (or other proof-search algorithm) will be detected only when the tactics are executed: either when they attempt to use a proof constructor with bad arguments, or when a proof witness fails the type-checker.

Static type systems (such as ML's) have the advantage over dynamic type systems (such as Lisp's) that many errors are detected much earlier, without needing to run the program on an adequate sample of test cases. The languages in which the above-described theorem provers are implemented – Standard ML, Caml, Lambda Prolog– all have static checking. But ML-style polymorphism is not strong enough to catch all programming errors.

We had experience in building a complicated tactical prover prototype (in Lambda Prolog) for a proof-carrying code application (Appel & Felty, 2000). We had a collection of complicated, ad-hoc tactics. As we maintained the prover, from time to time we found that it built invalid proofs; to debug, we had to do runtime tracing of appropriately stripped-down test cases to isolate the problem.

As we will show, using a dependently typed programming language can yield a partial correctness (i.e. soundness) guarantee for a theorem prover: if the implementation type-checks, then any proof (or subproof) that it builds will be valid. There is no total correctness (i.e. completeness) guarantee: that is, the prover might still infinite-loop or be incomplete in some other way, i.e. fail with a run-time exception (in ML) or backtracking failure (Prolog).

https://doi.org/10.1017/S0956796803004921 Published online by Cambridge University Press

The source code for all our examples can be found at www.cs.princeton.edu/~appel/prover/.

2 LF and Twelf

The logical framework LF (Harper *et al.*, 1993) allows the specification of logics, and implementations of LF such as Twelf (Pfenning & Schürmann, 1999) allow checking of proofs in those logics. Another view of LF/Twelf is that it is a higher-order dependently typed logic-programming language: Prolog, with higher-order abstract syntax (as in Lambda Prolog), well-scoped dynamic clauses, and a dependent type system. We shall make use of both views of LF/Twelf: we specify an object logic (e.g. first-order logic or higher-order logic), and we shall also do Prolog-like programming of the prover tactics.

The Twelf implementation of LF has (partial) type inference, proof search (i.e. Prolog-style backtracking), and constraint domains (e.g. the theory of the rational numbers). Twelf has a distinguished type *type*, the type of all types (and the type of logic-programming goals). A *constructor declaration* declares an axiom or inference rule of a logic, or a logic-programming data-constructor, or a logic-programming clause. A *definition* can be used to make a theorem, a lemma, or a defined function or predicate.[1]

An object logic. We shall use Twelf to write theorem provers. We begin by defining operators and axioms of an object logic: here we use first-order logic, which is encoded by the Twelf declarations in figure 1. Everything we do in this paper also works for higher-order and other logics; but we wish to simplify the presentation.

The declaration i : *type* declares the type i of individuals (over which the quantifiers range), and o : *type* declares the type of logical formulas (booleans). The constant *pf* is a dependent type constructor: for any formula A, $pf(A)$ is a type; we interpret this type to mean, “proofs of the formula A.” That is, if p has type $pf(\mathit{false\ imp\ true})$, then p must be a proof of *false imp true*.

The %use declaration brings in the (built-in) theory of the rational numbers, with constants 0, 1, 2, 3/2, 248/83, and so on, and operators $+$, $\times$, $>$, $\geqslant$. Though we don’t need the full power of the rationals – we use numbers only to index elements of our hypothesis list – this is Twelf’s preferred number system, so it’s simplest to just use it. We define a datatype constructor $\mathit{const} : \mathit{rational} \rightarrow i$ to inject rational constants into our logic’s element type.

We define infix operators *imp*, *and*, and *or* to construct formulas. The proof-constructor *and_i* (and-introduction) can be read as, “function taking a proof of A and returning (function taking a proof of B and returning a proof of A *and* B).” Thus if $p1 : pf(A)$ and $p2 : pf(B)$, then $\mathit{and\_i}\ p1\ p2 : pf(A\ \mathit{and}\ B)$.

The proof-constructor *imp_i* (implication-introduction) can be read as, “function taking (function from proof of A to proof of B) and returning proof of (A *imp* B).”

[1] Or even a logic-programming clause justified by a proof, though we won’t use that capability in this paper.

https://doi.org/10.1017/S0956796803004921 Published online by Cambridge University Press

```
i         :  type.
o         :  type.
pf        :  o → type.

%use inequality/rationals.
const     :  rational → i.

imp       :  o → o → o.      %infix right 10 imp.
imp_i     :  (pf A → pf B) → pf (A imp B).
imp_e     :  pf (A imp B) → pf A → pf B.

and       :  o → o → o.     %infix right 12 and.
and_i     :  pf A → pf B → pf(A and B).
and_e1    :  pf(A and B) → pf A.
and_e2    :  pf(A and B) → pf B.

or        :  o → o → o.     %infix right 11 or.
or_i1     :  pf A → pf(A or B).
or_i2     :  pf B → pf(A or B).
or_e      :  pf(A or B) → (pf A → pf C) → (pf B → pf C) → pf C.

forall    :  (i → o) → o.
forall_i  :  ({x : i} pf (A x)) → pf(forall A).
forall_e  :  pf(forall A) → {x : i} pf(A x).

exists    :  (i → o) → o.
exists_i  :  {x : i}pf(A x) → pf(exists A).
exists_e  :  pf(exists A) → ({x : i} pf(A x) → pf B) → pf B.

false     :  o.
false_e   :  pf false → pf A.
```

Fig. 1. First-order logic.

Twelf's function notation uses square brackets for lambda, thus ($[p]\ \mathit{and\_i}\ p\ p$) is a function with formal parameter p and result ($\mathit{and\_i}\ p\ p$). Alternately, we can read $\mathit{imp\_i}[p:\ \mathit{pf}\ A]\ Q(p)$ to mean, assuming A is true (with proof p), then the expression $Q(p)$ is a proof of B; thus $A\ \mathit{imp}\ B$.

In the following lemma, represented as a Twelf definition, we apply *imp_i* to this function to get the proof in the body of the definition.

$$\mathit{lemma}1\ :\ \mathit{pf}\ (A\ \mathit{imp}\ (A\ \mathit{and}\ A))\ =\ \mathit{imp\_i}\ ([p:\ \mathit{pf}\ A]\ \mathit{and\_i}\ p\ p).^{2}$$

As in most presentations of lambda-calculus, the lambda (square brackets) has a syntactic scope that extends as far as possible to the right; Twelf can reconstruct the type of the function argument; and our *and* binds tighter than *imp*; so we could

[2] Unbound capitalized variables are implictly universally quantified, so Twelf would internally reconstruct this definition to

$$\mathit{lemma}1\ :\ \{A:\ o\}\ \mathit{pf}\ (A\ \mathit{imp}\ (A\ \mathit{and}\ A))\ =\ [A:\ o]\ \mathit{imp\_i}\ ([p:\ \mathit{pf}\ A]\ \mathit{and\_i}\ p\ p).$$

where the curly braces construct dependent types: the type of *lemma*1 is, in effect, "function from formula (call it A) to proofs of $A\ \mathit{imp}\ (A\ \mathit{and}\ A)$."

also write

$$\mathit{lemma1}\ :\ \mathit{pf}\,(A\ \mathit{imp}\ A\ \mathit{and}\ A)\ =\ \mathit{imp\_i}\ [p]\ \mathit{and\_i}\ p\ p.$$

Using this style of definition and proof, we introduce some useful definitions and lemmas:

$$\begin{array}{lcl}
\mathit{not} & : & o \to o\ =\ [A]\ A\ \mathit{imp\ false}.\\
\mathit{not\_i} & : & (\mathit{pf}\ A \to \mathit{pf\ false}) \to \mathit{pf}\,(\mathit{not}\ A)\ =\ \mathit{imp\_i}.\\
\mathit{not\_e} & : & \mathit{pf}(\mathit{not}\ A) \to \mathit{pf}\ A \to \mathit{pf\ false}\ =\ \mathit{imp\_e}.\\
\mathit{true} & : & o\ =\ \mathit{not\ false}.\\
\mathit{true\_i} & : & \mathit{pf}(\mathit{true})\ =\ \mathit{not\_i}\ [p]\ p.
\end{array}$$

Notational definitions in Twelf are like type abbreviations in ML: the type-checker can freely expand them when type-checking. Furthermore, the type-checker's unifier uses rules of beta-eta equivalence. Thus, the proof of the true-introduction rule *true_i* must have type *pf*(*true*) which is equivalent (by definition) to *pf*(*not false*); the right-hand-side of *true_i* is *not_i* [*p*] *p* whose type is indeed *pf*(*not false*). Note that even though *not_i* is defined to be *imp_i*, it is really the special case where the *B* in *imp_i* is instantiated with *false*.

These definitions – including the proofs of the lemmas *not_i*,*not_e*,*true_i* – are type-checked by the system, so they can't be invalid. This means that we don't really need a prover at all; we could just write proofs by hand (as definitions) and check them in Twelf's type-checker; and in fact, such a method can be quite effective and useful (Appel, 2000).

However, we wish to automate: we will write a program to produce proofs semiautomatically or automatically, guided by tactical hints. Since Twelf's support for interactive I/O is minimal, in the prototype we do "interactive" tactical proving by editing proof-scripts.

3 A theorem prover using tactics and tacticals

Our prover manipulates *goals*, which are data structures of the form $h_1,\ldots,h_n \vdash h$, where each of the h_i is a hypothesis, represented as a formula with attached proof. For $h_1,\ldots,h_n$ we assume that the proof is already constructed. The conclusion h is also a formula with attached proof; typically we have not yet found the proof, so its "attached proof" is an uninstantiated logic variable.

The Twelf declarations for such data structures are as follows. *hyp* is the type of a single hypothesis, and *hyps* is a list of hypotheses:

$$\begin{array}{lcl}
\mathit{hyp} & : & \mathit{type}.\\
\mathit{hyps} & : & \mathit{type}.
\end{array}$$

An individual hypothesis is a pair of some formula A and a proof of that formula; we declare the nonassociative infix constructor *by* to construct such formula/proof

https://doi.org/10.1017/S0956796803004921 Published online by Cambridge University Press

pairs:

$$by : \underset{\text{argument 1}}{\{A : o\}} \quad \underset{\text{argument 2}}{pf(A)} \quad \to \quad \underset{\text{result}}{hyp.} \qquad \%\mathit{infix\ none}\ 5\ by.$$

This is a dependently typed constructor. Thus, (*true by true_i*) is well typed, but (*false by true_i*) is ill typed, even though *false* is a formula and *true_i* is a proof – it's the wrong type of proof.

To write *A by P* instead of *by A P*, we declare *by* as an infix operator (nonassociative, binding tightness 5) using the %*infix* declaration shown above.

To make hypothesis lists we declare two constructors for *hyps*, where our *cons* is an infix comma:

$$\begin{array}{lll} nil & : & hyps. \\ , & : & hyp \to hyps \to hyps. \qquad \%\mathit{infix\ right}\ 4,. \end{array}$$

Now we can declare the *goal* type with its infix constructor $\vdash$[3].

$$\begin{array}{llll} goal & : & type. & \\ \vdash & : & hyps \to hyp \to goal. & \%\mathit{infix\ none}\ 3\ \vdash. \\ \& & : & goal \to goal \to goal. & \%\mathit{infix\ right}\ 2\ \&. \\ allp & : & (pf\ A \to goal) \to goal. & \\ alli & : & (i \to goal) \to goal. & \\ tt & : & goal. & \end{array}$$

In addition to the basic goal $h_1,\ldots,h_n \vdash h$ we have compound goals $G_1 \& G_2$ to represent the case where the use of a tactic results in several subgoals (remaining proof obligations). The empty goal *tt* is the identity for & and indicates no remaining proof obligations. As we explain later, we need separate constructors *allp* and *alli* because Twelf is not a polymorphic language. This implementation of goals can be viewed as the Twelf version of a similar implementation in Lambda Prolog (Felty, 1993). The programs which manipulate them, in particular the tacticals and the *maptac* program below, are similar also. They do not make any essential use of dependent types, and thus do not contribute to the partial correctness of our tactics. It is mainly the type of the *by* constructor introduced above that is important for guaranteeing partial correctness of our tactics.

Tactics. A tactic is a procedure which takes a goal as input and returns subgoals that remain to be proven. We first show some simple tactics that implement the application of inference rules and lemmas, and later show some more complex tactics which perform some proof search. We first need the type *tac* of *tactic names,* and

[3] Identifiers in the real Twelf system must be written in ASCII, of course, so we use the symbol `|-` for $\vdash$.

then we define the names of some tactics:

$$\begin{array}{lcl}
\mathit{tac} & : & \mathit{type}. \\
\mathit{initial\_tac} & : & \mathit{tac}. \\
\mathit{and\_r\_tac} & : & \mathit{tac}. \\
\mathit{and\_l\_tac} & : & \mathit{rational} \to \mathit{tac}. \\
\mathit{imp\_r\_tac} & : & \mathit{tac}. \\
\mathit{imp\_l\_tac} & : & \mathit{rational} \to \mathit{tac}.
\end{array}$$

We define *tactic* as the interpreter relation for the logic program; that is, the expression $\mathit{tactic}\ T\ G_1\ G_2$ is a logic-programming goal that applies the tactic named T to the proof obligation G_1, resulting in remaining proof obligations G_2.

$$\mathit{tactic} : \mathit{tac} \to \mathit{goal} \to \mathit{goal} \to \mathit{type}.$$

Finally, we define clauses for the *tactic* relation. Generally, there are one or two clauses for each tactic-name. Examples are:

$$\begin{array}{ll}
t1 : & \mathit{tactic}\ \mathit{initial\_tac}\ (\mathit{Hs} \vdash A\ \mathit{by}\ P)\ \mathit{tt} \leftarrow \\
 & \qquad\qquad \mathit{nth\_item}\ N\ (A\ \mathit{by}\ P)\ \mathit{Hs}. \\
t2 : & \mathit{tactic}\ \mathit{and\_r\_tac}\ (\mathit{Hs} \vdash (A\ \mathit{and}\ B)\ \mathit{by}\ (\mathit{and\_i}\ P1\ P2)) \\
 & \quad (\mathit{Hs} \vdash A\ \mathit{by}\ P1\ \&\ \mathit{Hs} \vdash B\ \mathit{by}\ P2). \\
t3 : & \mathit{tactic}\ \mathit{imp\_r\_tac}\ (\mathit{Hs} \vdash (A\ \mathit{imp}\ B)\ \mathit{by}\ (\mathit{imp\_i}\ P1)) \\
 & \quad (\mathit{allp}\ [p2 : \mathit{pf}\ A](A\ \mathit{by}\ p2,\ \mathit{Hs} \vdash B\ \mathit{by}\ (P1\ p2))). \\
t9 : & \mathit{tactic}\ (\mathit{and\_l\_tac}\ N)\ (\mathit{Hs} \vdash C\ \mathit{by}\ P) \\
 & \quad ((A\ \mathit{by}\ (\mathit{and\_e1}\ Q)),\ (B\ \mathit{by}\ (\mathit{and\_e2}\ Q)),\ \mathit{Hs} \vdash C\ \mathit{by}\ P) \leftarrow \\
 & \quad \mathit{nth\_item}\ N\ ((A\ \mathit{and}\ B)\ \mathit{by}\ Q)\ \mathit{Hs}. \\
t11 : & \mathit{tactic}\ (\mathit{imp\_l\_tac}\ N)\ (\mathit{Hs} \vdash C\ \mathit{by}\ P) \\
 & \quad ((\mathit{Hs} \vdash A\ \mathit{by}\ P2)\ \&\ ((B\ \mathit{by}\ (\mathit{imp\_e}\ P1\ P2)),\ \mathit{Hs} \vdash C\ \mathit{by}\ P)) \leftarrow \\
 & \quad \mathit{nth\_item}\ N\ ((A\ \mathit{imp}\ B)\ \mathit{by}\ P1)\ \mathit{Hs}.
\end{array}$$

The lines $t1, t2, \ldots$ can be understood as logic-programming clauses, where ← is used instead of the Prolog or Lambda Prolog `:-`. Thus, the rule $t1$ might be written in Lambda Prolog as

```
tactic initial_tac (Hs |- (A by P)) tt :-
    nth_item N (A by P) Hs.
```

where the data constructors `|-` and `by` are infix (of course, in Lambda Prolog the type-checker can't check soundness of the tactic).

The operational interpretation of a Prolog clause H `:-` G_1`;` G_2`;` G_3 or a Twelf clause $H \leftarrow G_1 \leftarrow G_2 \leftarrow G_2$ is, first match the head H against the current goal. If that matches, try and satisfy subgoal G_1; if that matches, satisfy subgoal G_2, and so on. Twelf, like Prolog, uses backtracking (so that if G_2 fails, then a different way of satisfying G_1 is tried, and so on).

The supporting clauses for *nth_item N H Hs* are straightforward (typed) Prolog, and define the relation that the *N*th item of *Hs* is precisely *H*:

$$\begin{array}{lll} \mathit{nth\_item} & : & \mathit{rational} \to \mathit{hyp} \to \mathit{hyps} \to \mathit{type}. \\ \mathit{nth\_item}1 & : & \mathit{nth\_item}\ 1\ H1\ (H1,\ Hs). \\ \mathit{nth\_item}N & : & \mathit{nth\_item}\ N\ H1\ (H2,\ Hs)\ \leftarrow\ \mathit{nth\_item}\ (N-1)\ H1\ Hs. \end{array}$$

Thus, *initial_tac* matches a goal $Hs \vdash A\ by\ P$ if there exists an *N* such that the hypothesis *A by P* is the *N*th item of *Hs* (in Isabelle this is called *assume_tac*).

We can let Prolog backtracking find the right *N* for *initial_tac* because the subgoals are trivial, but for *and_l_tac* it would be unwise to rely on this, because *and_l_tac* has nontrivial subgoals. Therefore the user must supply a number when using *and_l_tac*, but has the option of supplying a Prolog unification variable, which causes *nth_item* to do a backtracking search for an assumption of the form *A and B*.

The tactic implementation of most of the right introduction rules of our object logic is straightforward. The input goal contains the conclusion paired with its proof, and the output goal contains the hypotheses paired with their proofs. If there is more than one subgoal, they are connected by &, as in *and_r_tac*. Rules which use nested implication or quantification in Twelf such as *imp_i* and *forall_i* in Figure 1 must use one of the *all* goal constructors in their tactic implementations. For example, the argument to *imp_i* is a function from proofs of *A* to proofs of *B*. In the tactic version (*t*3 above), the use of *allp* introduces a bound variable *p*2 to represent an arbitrary proof of *A* which gets paired with *A* and added to the assumption list *Hs* of the subgoal.

The tactics for the left introduction rules are implemented so that they perform forward proof from hypotheses. An argument is given to indicate the position in the hypothesis list of the hypothesis to which the rule should be applied. The partial proofs are constructed and added to the hypothesis lists of the subgoals.

For each of the left introduction rules, we provide a second version of the tactic which removes the hypothesis to which the specified rule is applied when forming the subgoal. For example, for and-elimination, we have:

$$\begin{array}{ll} t10: & \mathit{tactic}\ (\mathit{and\_l\_tacR}\ N)\ (Hs1\ \vdash\ C\ \mathit{by}\ P) \\ & \qquad ((A\ \mathit{by}\ (\mathit{and\_e}1\ Q)),\ (B\ \mathit{by}\ (\mathit{and\_e}2\ Q)),\ Hs2\ \vdash\ C\ \mathit{by}\ P)\ \leftarrow \\ & \quad \mathit{nth\_and\_rest}\ N\ ((A\ \mathit{and}\ B)\ \mathit{by}\ Q)\ Hs1\ Hs2. \end{array}$$

where *nth_and_rest* is a logic-programming predicate which finds the *N*th formula in *Hs*1 and returns the set of hypotheses *Hs*2 with the *N*th one removed. Such tactics are useful in writing automated proof search procedures so that they can avoid repeatedly applying the same rule to the same hypothesis.

More tactics. Using these general principles, it's easy to implement a large variety of tactics. Here we show three more:

$$\begin{array}{ll} \mathit{forall\_r\_tac}: & \mathit{tac}. \\ \mathit{forall\_l\_tac}: & \mathit{rational} \to \mathit{tac}. \\ \mathit{resolve2\_tac}: & (\mathit{pf}\ A1 \to \mathit{pf}\ A2 \to \mathit{pf}\ B) \to \mathit{tac}. \end{array}$$

https://doi.org/10.1017/S0956796803004921 Published online by Cambridge University Press

$$\begin{aligned} t7 :\quad & \mathit{tactic\ forall\_r\_tac}\ (\Gamma\ \vdash\ (\mathit{forall}\ A)\ \mathit{by}\ (\mathit{forall\_i}\ P)) \\ & \qquad (\mathit{alli}\ [t : i](\Gamma\ \vdash\ (A\ t)\ \mathit{by}\ (P\ t))). \end{aligned}$$

$$\begin{aligned} t17 :\quad & \mathit{tactic}\ (\mathit{forall\_l\_tac}\ N)\ (\Gamma\ \vdash\ C\ \mathit{by}\ P) \\ & \qquad\qquad (((A\ X)\ \mathit{by}\ (\mathit{forall\_e}\ Q\ X)) ; \Gamma\ \vdash\ C\ \mathit{by}\ P)\ \leftarrow \\ & \qquad \mathit{nth\_item}\ N\ ((\mathit{forall}\ A)\ \mathit{by}\ Q)\ \Gamma. \end{aligned}$$

$$\begin{aligned} t25 :\quad & \mathit{tactic}\ (\mathit{resolve2\_tac}\ (\mathit{Thm} :\ \mathit{pf}\ A1 \to \mathit{pf}\ A2 \to \mathit{pf}\ B)) \\ & \qquad (\Gamma\ \vdash\ B\ \mathit{by}\ (\mathit{Thm}\ P1\ P2)) \\ & \qquad (\Gamma\ \vdash\ A1\ \mathit{by}\ P1\ \ \&\ \ \Gamma\ \vdash\ A2\ \mathit{by}\ P2). \end{aligned}$$

To prove a universally quantified formula $\forall x{:}A(x)$, *forall_r_tac* introduces an *alli* goal; then clause *m*4 (shown below) will dynamically create an atom of type *i*, so that the subgoal, in effect, is to prove A with the new atom substituted for x. The substitution is handled entirely by the Twelf metalogic (the same would be true in Lambda Prolog).

To make use of a universally quantified hypothesis, *forall_l_tac* uses the argument N to select the Nth hypothesis from the assumptions, which must be of the form *forall* A (equivalently, *forall* $[x]\ A(x)$). A logic variable X is introduced to instantiate the bound variable in A. It can later be unified with a term that is needed to complete the proof. Then $A\ X$ is unified with the hypothesis in the goal formula; although this is higher-order unification (which is undecidable in general), extensive experience with the use of similar tactics in Lambda Prolog has found them to work fine in practice. We can also write a version of this tactic that allows the user to provide the instantiation term X at the time the tactic is applied. We do this by adding X to the first argument as follows:

$$\mathit{forall\_l\_tacx} :\ \mathit{rational} \to i \to \mathit{tac}.$$

$$\begin{aligned} t17x :\quad & \mathit{tactic}\ (\mathit{forall\_l\_tacx}\ N\ X)\ (\Gamma\ \vdash\ C\ \mathit{by}\ P) \\ & \qquad\qquad (((A\ X)\ \mathit{by}\ (\mathit{forall\_e}\ Q\ X)) ; \Gamma\ \vdash\ C\ \mathit{by}\ P)\ \leftarrow \\ & \qquad \mathit{nth\_item}\ N\ ((\mathit{forall}\ A)\ \mathit{by}\ Q)\ \Gamma. \end{aligned}$$

We have also shown an example of a resolution tactic. Given some theorem T of the form, $\mathit{pf}(A_1) \to \mathit{pf}(A_2) \to \mathit{pf}(B)$, the tactic *resolve2_tacT* matches a goal B and produces subgoals A_1 and A_2. A minor disadvantage of doing this in a well typed way is that we need a different tactic for 2-premise theorems than for 3-premise theorems, and so on. Note that the user need not type in a proof term for the *Thm* argument directly. Instead, the name of a previously defined Twelf declaration which expresses a lemma can be given, as long as it has the right type. By Twelf definition expansion, this name is equivalent to the term it abbreviates.

Tacticals. Tacticals implement basic control mechanisms which allow simple tactics to be combined into more complex ones, and can be used as a programming language to implement search procedures. Most tacticals assume the input goal is a basic goal (constructed using $\vdash$ in our prover). In the logic programming setting, we first implement a *maptac* tactical which applies tactics to compound goals, reducing them to basic goals before passing them on to other tacticals and tactics.

```
maptac :  tac → goal → goal → type.

m1 :      maptac T tt tt.
m2 :      maptac T (InG1 & InG2) (OutG1 & OutG2) ←
               maptac T InG1 OutG1 ← maptac T InG2 OutG2.
m3 :      maptac T (allp InG) (allp OutG) ←
               {p} maptac T (InG p) (OutG p).
m4 :      maptac T (alli InG) (alli OutG) ←
               {t} maptac T (InG t) (OutG t).
m5 :      maptac T (Hs ⊢ A by P) OutG ←
               tactic T (Hs ⊢ A by P) OutG.
```

This tactical reduces the goal to subgoals in a manner consistent with the meaning of the top-level goal constructor. In the clauses for the *all* constructors, the quantification within goals is transferred to quantification in Twelf. For example, *allp* quantifies over proofs in the object logic; in the *m*3 clause, *p* is introduced as an arbitrary proof to replace the bound variable in *InG*. After completion of the Twelf subgoal, *OutG* is also an abstraction over *p*.

Since *maptac* has the same type as *tactic*, we could have dispensed with *maptac* and written *m*1*;…;m*4 as clauses for *tactic*; but this would allow the user less control of how and when the tactics are applied.

Some common tacticals found in most tactic-style theorem provers are implemented in Twelf with the following clauses.

```
idtac :      tac.
then :       tac → tac → tac.   %infix left 2 then.
orelse :     tac → tac → tac.   %infix left 2 orelse.
repeat :     tac → tac.
try :        tac → tac.
complete :   tac → tac.

tactical1 :  tactic idtac G G.
tactical2 :  tactic (T1 then T2) InG OutG ←
                 tactic T1 InG MidG ← maptac T2 MidG OutG.
tactical3 :  tactic (T1 orelse T2) InG OutG ← tactic T1 InG OutG.
tactical4 :  tactic (T1 orelse T2) InG OutG ← tactic T2 InG OutG.
tactical5 :  tactic (repeat T) InG OutG ←
                 tactic ((T then (repeat T)) orelse idtac) InG OutG.
tactical6 :  tactic (try T) InG OutG ← tactic (T orelse idtac) InG OutG.
tactical7 :  tactic (complete T) InG tt ←
                 tactic T InG OutG ← goalreduce OutG tt.
```

The *idtac* tactical returns the goal unchanged and is used mainly in programming search strategies for ending a series of multiple proof steps. The *then* tactical performs the composition of tactics. The *orelse* tactical is also useful in programming search

https://doi.org/10.1017/S0956796803004921 Published online by Cambridge University Press

strategies and allows choice of tactics. The *repeat* tactical repeatedly applies a tactic until it can no longer be applied and is defined in terms of the others. The *try* tactical prevents failure of the given argument tactic by using *idtac* when tactic T fails. Finally the *complete* tactical tries to completely solve the given goal. It uses *goalreduce* (not shown) to simplify compound goal expressions by removing occurrences of *tt* from them. For example, applying multiple tactics could result in goal structures such as (*allp* ([x] *tt* & *tt*)) whose only subgoals are *tt* and so should reduce to *tt*.

4 A more intricate tactic

An important property of a tactical prover is that it is *extensible*, so that its users can write their own tactics. It is in the checking of user-defined tactics that the dependent type system is particularly useful. To illustrate, we will show a specialized tactic of the kind that some user might write.

Suppose we have a sum-of-products assertion,

$$\begin{aligned} C = {} & (A_{11} \wedge A_{21} \wedge A_{31} \wedge \top) \vee \\ & (A_{12} \wedge A_{22} \wedge \top) \vee \\ & (A_{13} \wedge A_{23} \wedge A_{33} \wedge A_{43} \wedge \top) \vee \\ & \bot \end{aligned}$$

and we want to prove C implies D, where we know $A_{i1} \vdash D$, $A_{i2} \vdash D$, $A_{i3} \vdash D$, for a particular i. To handle this we can write a tactic *sumprod*(i).

This kind of situation comes up, for example, in proving properties of a program that fetches from an ML-style sum-of-products datatype. Suppose some value x belongs to an ML datatype that has three constructors (disjuncts), which take values that are all records (of 3 elements, 2 elements, and 4 elements, respectively). We would like to fetch and use the second record field even before doing the case-analysis that tells us which disjunct applies. To do this "hoist" operation, we need to prove that the second field exists (in each disjunct) and has the right properties. The *sumprod* tactic will be useful in such proofs. Clearly, it's a very specialized situation – therefore this tactic will be user-defined, not provided by default.

We start with two preliminary lemmas. The specialized subtactics of *sumprod* will apply these specialized lemmas:

$$\begin{aligned} \mathit{or\_imp} \;:\; & \mathit{pf}\,(A\ \mathit{imp}\ C) \to \mathit{pf}\,(B\ \mathit{imp}\ C) \to \mathit{pf}\,((A\ \mathit{or}\ B)\ \mathit{imp}\ C) \ = \\ & [p1\ :\mathit{pf}\,(A\ \mathit{imp}\ C)]\ [p2\ :\mathit{pf}\,(B\ \mathit{imp}\ C)] \\ & \mathit{imp\_i}\ [p3 : \mathit{pf}\,(A\ \mathit{or}\ B)] \\ & \mathit{or\_e}\ p3\ ([p4 : \mathit{pf}\ A]\ \mathit{imp\_e}\ p1\ p4)\ ([p5\ :\mathit{pf}\ B]\ \mathit{imp\_e}\ p2\ p5). \end{aligned}$$

$$\begin{aligned} \mathit{and\_imp}\!: \;& \mathit{pf}(B\ \mathit{imp}\ D) \to \mathit{pf}(A\ \mathit{and}\ B\ \mathit{imp}\ D)\ = \\ & [p1\ : \mathit{pf}(B\ \mathit{imp}\ D)]\ \mathit{imp\_i}\ [p2 : \mathit{pf}\,(A\ \mathit{and}\ B)]\ \mathit{imp\_e}\ p1\ (\mathit{and\_e2}\ p2). \end{aligned}$$

We start with an auxiliary tactic *prodn*(j) that converts the goal $Hs \vdash (A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge \top) \rightarrow D$ to the goal $A_j, Hs \vdash D$.

https://doi.org/10.1017/S0956796803004921 Published online by Cambridge University Press

$$\begin{array}{lll}
\mathit{prodn}: & \mathit{rational} \rightarrow \mathit{tac}. & \\
t136 & : & \mathit{tactic}\,(\mathit{prodn}\ 1)\ (\mathit{Hs}\ \vdash\ \mathit{A\ and\ As\ imp\ D\ by}\ (\mathit{imp\_i}\ [p]\ P\,(\mathit{and\_e}1\ p))) \\
 & & \qquad\qquad (\mathit{allp}\ [p]\ (\mathit{A\ by\ p},\ \mathit{Hs}\ \vdash\ \mathit{D\ by\ P\ p})). \\
t137 & : & \mathit{tactic}\,(\mathit{prodn}\ N)\ (\mathit{Hs}\ \vdash\ \mathit{A\ and\ As\ imp\ D\ by\ and\_imp\ P})\ G\ \leftarrow \\
 & & \quad \mathit{tactic}\,(\mathit{prodn}\ (N\ -\ 1))\ (\mathit{Hs}\ \vdash\ \mathit{As\ imp\ D\ by\ P})\ G.
\end{array}$$

Finally, the tactic *sumprod*(i) transforms the goal $Hs \vdash (\bigvee_i \bigwedge_j A_{ij}) \rightarrow D$ to the goal $(A_{i1}, Hs \vdash D) \& \ldots \& (A_{in}, Hs \vdash D)$:

$$\begin{array}{lll}
\mathit{sumprod}: & \mathit{rational} \rightarrow \mathit{tac}. & \\
t134 & : & \mathit{tactic}\,(\mathit{sumprod}\ N)\ (\mathit{Hs}\ \vdash\ \mathit{false\ imp\ D\ by}\ (\mathit{imp\_i\ false\_e}))\ \mathit{tt}. \\
t135 & : & \mathit{tactic}\,(\mathit{sumprod}\ N) \\
 & & \quad (\mathit{Hs}\ \vdash\ (\mathit{A\ or\ As})\ \mathit{imp\ D\ by}\ (\mathit{or\_imp}\ P1\ \ P2))\ (G1\ \&\ G2)\ \leftarrow \\
 & & \quad\ \mathit{tactic}\,(\mathit{prodn}\ N)\ (\mathit{Hs}\ \vdash\ \mathit{A\ imp\ D\ by}\ P1)\ G1\ \leftarrow \\
 & & \quad\ \mathit{tactic}\,(\mathit{sumprod}\ N)\ (\mathit{Hs}\ \vdash\ \mathit{As\ imp\ D\ by}\ P2)\ G2.
\end{array}$$

To see how the dependent type system ensures that we got this right, let's examine the typechecking of rule $t135$. As reconstructed by Twelf's typechecker, we have,

```
t135 :
  {N:rational} {Hs:hyps} {As:o} {D:o} {P2:pf (As imp D)} {G2:goal} {A:o}
    {P1:pf (A imp D)} {G1:goal}
    tactic (sumprod N) (Hs |- As imp D by P2) G2
      -> tactic (prodn N) (Hs |- A imp D by P1) G1
      -> tactic (sumprod N) (Hs |- A or As imp D by or_imp P1 P2) (G1 & G2).
```

Here we have explicit metalevel quantification (using curly braces) of all the implicitly quantified logical variables N, Hs, As, D, etc. The type of $P1$ was inferred from the expression *A imp D by P*1: it must be *pf*(*A imp D*). Therefore, the use of $P1$ in the expression *or_imp P*1 *P*2 typechecks.

Yet, suppose we had mistakenly written the rule $t135$ with

$$\mathit{A\ or\ As\ imp\ D\ by\ and\_imp}\ P2.$$

Then this rule wouldn't type-check, and Twelf would report the error,

```
Type mismatch
Expected: pf ('A or 'As imp 'D)
Found:    pf (X1 and 'As imp 'D)
```

When writing tactics such as this (but quite a bit messier) in Lambda Prolog, we found that mismatches between tactics and the lemmas that they apply were one of the two common sources of errors in the prover; such errors do not impede us in Twelf. The other kind of error – incompleteness via infinite loops or backtracking failure – continues to be bothersome, of course: dependent types do not save us there.

5 Union-Find

Not only tactical provers, but also other decision procedures can be dependently typed to ensure partial correctness. For example, in decision procedures for equality, the standard efficient union-find algorithm with path compression (Aho *et al.*, 1974) is often used to represent equivalence classes.

For each equivalence class, the algorithm maintains a canonical representative. As new equalities are learned (from some other source), the algorithm is instructed (by a *union a b* command) to merge the two equivalence classes to which a and b belong. To query the data structure, the *find a B* command seeks the canonical representative of the class to which a belongs, and unifies it with B. In the context of our theorem prover, *find* must also produce a proof that $a = b$.

We have implemented a union-find prover in Twelf. Assuming that the logic-programming engine efficiently indexes atomic dynamic clauses[4], it should run in $O(N\,\alpha(N))$, where $\alpha(N)$ is the inverse Ackermann function.

In our example, we add an equality primitive == to the logic, along with some axioms. Union-find will maintain and query canonical representatives of equivalence classes:

$$\begin{array}{l}
==: i \rightarrow i \rightarrow o. \quad \%\mathit{infix\ none}\ 20\ == .\\
\mathit{refl}: \ \mathit{pf}(A == A).\\
\mathit{symm}: \ \mathit{pf}(A == B) \rightarrow \mathit{pf}(B == A).\\
\mathit{trans}: \ \mathit{pf}(A == B) \rightarrow \mathit{pf}(B == C) \rightarrow \mathit{pf}(A == C).
\end{array}$$

Some of the important constructors and predicates used in this example are declared as follows.

$$\begin{array}{l}
\vdash: \ \mathit{hyps} \rightarrow \mathit{pf}\ A \rightarrow \mathit{goal}. \quad \%\mathit{infix\ none}\ 3\ \vdash .\\
\mathit{union}: \ \mathit{pf}(X == Y) \rightarrow \mathit{hyps}.\\
\mathit{find}: \ \{x\}\{y\}\mathit{pf}(x == y) \rightarrow \mathit{hyps}.\\
\mathit{canonical}: \ i \rightarrow \mathit{type}.
\end{array}$$

Assume we have a function f and some primitive equality facts:

$$f: \ \mathit{rational} \rightarrow i.$$

$$\begin{array}{l}
u35: \ \mathit{pf}(f\,3 \ == \ f\,5).\\
u79: \ \mathit{pf}(f\,7 \ == \ f\,9).\\
u75: \ \mathit{pf}(f\,7 \ == \ f\,5).
\end{array}$$

$$\begin{array}{l}
\mathit{find2}: \mathit{pf}(A == B) \ \rightarrow \ \mathit{pf}(C == B) \ \rightarrow \ \mathit{pf}(A == C) \ =\\
\quad [pAB][pCB] \ \mathit{trans}\ pAB\ (\mathit{symm}\ pCB).
\end{array}$$

A typical query that our union-find can answer is,

$$\begin{array}{r}
\mathit{union}\ u35,\ \mathit{union}\ u79,\ \mathit{union}\ u75,\ \mathit{find}\,(f\,9)\ X\ P9,\ \mathit{find}\,(f\,3)\ X\ P3,\ \mathit{nil}\\
\vdash \ f\,9 \ == \ f\,3\ \ \mathit{by}\ \mathit{find2}\ P9\ P3.
\end{array}$$

[4] Dynamic clauses will be explained in this section. Twelf does not index dynamic clauses, so a real test of our program's efficiency has not yet been performed.

https://doi.org/10.1017/S0956796803004921 Published online by Cambridge University Press

In this prover, the "hypotheses" to the left of the turnstile $\vdash$ are treated as commands to the union-find engine. Associated with each command is a proof: *union* P (where P is a proof of $A == B$) is a command to union the sets to which A and B belong. *find* $X\ Y\ P$ is a command to find the canonical representative of X, unify it with Y, and construct a proof that $X == Y$; this proof is then unified with P. Thus, by the time *nil* is reached, the proof to the right of the turnstile in our example, *find*2 $P2\ P3$, must be a proof of $f9 == f3$.

How could such a query fail? In our example, the only possible point is where the command *find* $(f3)\ X\ P3$ is executed: here, X has already been instantiated to the canonical representative of $f9$, so if that is not the same as the canonical representative of $f3$, the *find* command will fail and backtrack. In this example, such failure does not occur.

Our program introduces dynamic clauses of the form *canon* $X\,Z\,Pxz$ to indicate that Z is the canonical representative of X, with proof Pxz:

$$\mathit{canon} : \ \{x:i\}\{y:i\}\ \mathit{pf}(x == y) \rightarrow \mathit{type}.$$

That is, these clauses of the Prolog program will be created at runtime by the execution of other clauses. Standard Prolog has **assert** and **retract** to add and delete clauses to/from the fact database; both LambdaProlog and Twelf have a dynamically scoped version of this feature, in which dynamically added clauses are automatically removed when the goals containing them complete successfully, or when backtracking occurs. A Twelf clause such as

$$c : \ \mathit{expr}1 \ \leftarrow \ \{d : \ \mathit{expr}2\}\ \mathit{expr}3.$$

would operate as follows: if the top-level goal matches *expr*1, then the subgoal becomes $\{d : \ \mathit{expr}2\}\ \mathit{expr}3$; to satisfy this subgoal, first the clause $d : \ \mathit{expr}2$ is added to the fact database, then the subgoal *expr*3 is tried. Once *expr*3 succeeds or fails, the dynamic clause $d : \ \mathit{expr}2$ is removed.

Our program has 16 clauses and 13 constructor declarations. Instead of showing the whole program, we will show just one clause to illustrate the use of dependent types. The following clause "executes" a command *find* $X\,Y\,P$ in the case that X maps in exactly two steps to Y; in this case, path-compression is performed:

$$\begin{array}{ll} \mathit{find\_tac2} : & \mathit{find}\ X\ Y\ P,\ Hs \vdash \ H \ \leftarrow \\ & \mathit{canon}\ X\ Z\ Pxz \ \leftarrow \\ & \mathit{canon}\ Z\ Y\ Pzy \ \leftarrow \\ & \mathit{canonical}\ Y \ \leftarrow\ !\ \leftarrow \\ & \{d : \ \mathit{canon}\ X\ Y\ (\mathit{trans}\ Pxz\ Pzy)\}\ Hs \ \vdash \ H. \end{array}$$

The first line matches the find command; lines 2 and 3 match the case that X links to Y in two steps, with proofs Pxz and Pzy respectively; line 4 checks that Y is its own canonical representative. Then there is a Prolog "cut" (!), to prevent other interpretations of the *find* command from matching[5]. Then a new atomic clause is

[5] We are using a version of Twelf with "cut"; the standard distribution does not have this operation.

https://doi.org/10.1017/S0956796803004921 Published online by Cambridge University Press

added to the global database, stating that Y is the canonical representative of X with proof *trans Pxz Pzy*; finally, the remaining command-list *Hs* is executed. The old clause *canon X Z Pxz* is still there, but by careful use of cuts, the algorithm will never have occasion to use it.

When *find_tac*2 adds a new clause to the global database, the dependent type of the *canon* constructor ensures that it must be with a valid proof. When a proof P is returned after a set of commands $Hs \vdash A\ by\ P$, the dependent type of $\vdash$ ensures that it proves the theorem that is claimed. The correctness of *find_tac*2 and similar clauses is guaranteed statically.

6 Related work

Using dependent types in proofs was not possible in the corresponding Lambda Prolog version of our tactic-style theorem prover. Lambda Prolog, however, has polymorphic types, which Twelf does not, and these types provide some advantages in a Lambda Prolog implementation of tactics and tacticals. For example, in Lambda Prolog, only one version of the goal constructor for universal quantification is needed:

$$all \quad : \quad (A \rightarrow goal) \rightarrow goal.$$

where A is a type variable that can be instantiated with any type. Thus, the implementation of the goal constructors and tacticals does not have to change when we change object logics. In contrast, in Twelf, one *all* constructor is needed for each type that needs to be quantified. Twelf also does not allow quantification over predicates. In Lambda Prolog, tactics can be implemented as predicates taking two goals as arguments, which means that tacticals would have predicate arguments. To illustrate, if this were possible in Twelf, there would no longer be a need for the *tactic* constructor and the type $goal \rightarrow goal \rightarrow type$ would become the definition of the type *tac*. Some of the code would look like:

$$\begin{array}{lll}
tac & = & goal \rightarrow goal \rightarrow type. \\
t1: & & initial\_tac\ (Hs \vdash\ A\ by\ P)\ tt\ \leftarrow \\
 & & \qquad\qquad nth\_item\ N\ (A\ by\ P)\ Hs. \\
tactical2: & & then\ T1\ \ T2\ \ InG\ \ OutG\ \leftarrow \\
 & & \qquad T1\ \ InG\ \ MidG\ \leftarrow\ T2\ \ MidG\ \ OutG.
\end{array}$$

Pollack (1995) discusses the use of dependent types in LCF-style provers to avoid the need for validations. As a first step, a modification of the unforgeable abstract data type *theorem* is presented. The new data type makes the structure of the theorem explicit in the ML type, resulting in a more informative type. Then, a more expressive metalanguage with dependent types is proposed. When taking this step, the notion of tactic is modified; a tactic in this setting becomes the statement of a derived or admissible rule along with its proof in the LEGO system (Pollack, 1994). Applying the tactic means applying the new rule as a lemma. Programming decision procedures for proving subgoals is also mentioned, but example programs are not given.

McBride (2001) presents an implementation of first-order unification using a dependently typed functional language derived from the LEGO system. The language is a strongly normalizing type theory, so he is able to establish termination. Bove (1999) also programs unification in a dependently typed functional language. She uses Martin-Löf's type theory as a programming language and works within the the ALF system (Altenkirch *et al.*, 1994). She also establishes termination. In addition, she provides a methodology for extracting a Haskell program from the type theory version. It would be interesting to compare these programs to a dependently typed logic-programming implementation of the same algorithm.

7 Conclusion

We have shown how dependent types can guarantee partial correctness of tactics in a tactic-style theorem prover written in Twelf. We have also shown that other proof strategies such as decision procedures can benefit similarly from the use of dependent types. In both of these examples, the fact that object-level proofs were constructed and returned as a result of proof search was a crucial element of the program. By using dependent types to represent such proofs, it is not possible to write tactics or other proof procedures that construct proofs that will not check when submitted to a proof checker.

Both Coq and Twelf contain dependently typed languages intended for describing object-logic terms. The designers of these systems didn't really intend that large-scale programs written in these "little" languages would be executed within Coq or Twelf. We have demonstrated that there's a significant software-engineering advantage to using the little language in Twelf instead of programming in ML, which is the surrounding implementation's language. The same demonstration could probably have been done using Coq's object language, a dependently typed functional language (as contrasted with Twelf's dependently typed Prolog-like language).

Although the tactical prover discussed in this paper is just a prototype, we are confident that these techniques will scale to full-size provers and decision procedures. We have used similar techniques in other dependently typed proof-manipulation programs in Twelf, and the dependent types assist, not impede, program development.

References

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974) *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

Altenkirch, T., Gaspes, V., Nordström, B. and von Sydow, B. (1994) *A user's guide to ALF.* Technical report, Chalmers University of Technology, Sweden.

Appel, A. W. (2000) *Hints on proving theorems in Twelf.* www.cs.princeton.edu/˜appel/twelf-tutorial.

Appel, A. W. and Felty, A. P. (2000) A semantic model of types and machine instructions for proof-carrying code. *POPL '00: 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 243–253. ACM Press.

Barras, B. *et al.* (1998) *The Coq Proof Assistant reference manual.* Technical report, INRIA.

https://doi.org/10.1017/S0956796803004921 Published online by Cambridge University Press

Bove, A. (1999) *Programming in Martin-Löf type theory: Unification, a non-trivial example.* Licentiate Thesis, Chalmers University of Technology and Göteborg University.

Felty, A. (1993) Implementing tactics and tacticals in a higher-order logic programming language. *J. Automated Reasoning*, **11**(1), 43–81.

Gordon, M. J. C. and Melham, T. F. (1993) *Introduction to HOL – A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press.

Gordon, M. J., Milner, R. and Wadsworth, C. P. (1979) *Edinburgh LCF: A mechanised logic of computation: Lecture Notes in Computer Science 78.* Springer-Verlag.

Harper, R., Honsell, F. and Plotkin, G. (1993) A framework for defining logics. *J. ACM*, **40**(1), 143–184.

McBride, C. (2001) First-order unification by structural recursion. *J. Functional Programming.* To appear.

Nadathur, G. and Miller, D. (1988) An overview of lambda prolog. In: Bowen, K. and Kowalski, R. (editors), *Fifth International Conference and Symposium on Logic Programming.* MIT Press.

Paulson, L. C. (1994) *Isabelle: A generic theorem prover: Lecture Notes in Computer Science 828.* Springer-Verlag.

Pfenning, F. and Schürmann, C. (1999) System description: Twelf — a meta-logical framework for deductive systems. *16th International Conference on Automated Deduction.* Springer-Verlag.

Pollack, R. (1994) *The theory of LEGO: A proof checker for the extended calculus of constructions.* PhD thesis, University of Edinburgh.

Pollack, R. (1995) On extensibility of proof checkers. In: Dybjer, P., Nordstrom, B. and Smith, J. (editors), *Types for Proofs and Programs: International Workshop Types'94: Lecture Notes in Computer Science 996.* Springer-Verlag.

https://doi.org/10.1017/S0956796803004921 Published online by Cambridge University Press