αlean*TAP*: A Declarative Theorem Prover for First-Order Classical Logic

Joseph P. Near[**], William E. Byrd, and Daniel P. Friedman

Indiana University, Bloomington, IN 47405
{jnear,webyrd,dfried}@cs.indiana.edu

Abstract. We present αlean*TAP*, a declarative tableau-based theorem prover written as a pure relation. Like lean*TAP*, on which it is based, αlean*TAP* can prove ground theorems in first-order classical logic. Since it is declarative, αlean*TAP* *generates* theorems and accepts non-ground theorems and proofs. The lack of mode restrictions also allows the user to provide guidance in proving complex theorems and to ask the prover to instantiate non-ground parts of theorems. We present a complete implementation of αlean*TAP*, beginning with a translation of lean*TAP* into αKanren, an embedding of nominal logic programming in Scheme. We then show how to use a combination of tagging and nominal unification to eliminate the impure operators inherited from lean*TAP*, resulting in a purely declarative theorem prover.

1 Introduction

We present a declarative theorem prover for first-order classical logic. We call this prover αlean*TAP*, since it is based on the lean*TAP* [1] prover and written in αKanren [2]. Our prover is a pure relation and has no mode restrictions [3]; given a logic variable as the theorem to be proved, αlean*TAP* *generates* valid theorems.

lean*TAP* is a lean tableau-based theorem prover for first-order logic due to Beckert and Posegga [1]. Written in Prolog, it is extremely concise and is capable of a high rate of inference. lean*TAP* uses Prolog's cut (`!`) in three of its five clauses in order to avoid nondeterminism, and uses `copy_term/2` to make copies of universally quantified formulas. Although Beckert and Posegga take advantage of Prolog's unification and backtracking features, their use of the impure cut and `copy_term/2` makes lean*TAP* non-declarative.

We show how to eliminate these impure operators from lean*TAP*. To eliminate the use of Prolog's cut, we introduce a tagging scheme that makes our formulas unambiguous. To eliminate the use of `copy_term/2`, we use substitution instead of copying terms. Universally quantified formulas are used as templates, rather than instantiated directly; instead of representing universally quantified variables with logic variables, we use the noms of nominal logic [4]. We then use nominal unification [5] to write a substitution relation that replaces quantified variables with logic variables, leaving the original template untouched.

[**] Now at the Massachusetts Institute of Technology: jnear@csail.mit.edu

The resulting declarative theorem prover is interesting for two reasons. First, because of the technique used to arrive at its definition: we use declarative substitution rather than `copy_term/2`. To our knowledge, there is no method for copying arbitrary terms declaratively. Our solution is not completely general but is useful when a term is used as a template for copying, as in the case of lean*TAP*. Second, because of the flexibility of the prover itself: αlean*TAP* is capable of instantiating non-ground theorems during the proof process, and accepts non-ground *proofs*, as well. Whereas lean*TAP* is fully automated and either succeeds or fails to prove a given theorem, αlean*TAP* can accept guidance from the user in the form of a partially-instantiated proof, regardless of whether the theorem is ground.

We present an implementation of αlean*TAP*, demonstrating our technique for eliminating cuts and `copy_term/2` from lean*TAP*. Our implementation demonstrates our contributions: first, it illustrates a method for eliminating common impure operators, and demonstrates the use of nominal logic for representing formulas in first-order logic; second, it shows that the tableau process can be represented as a relation between formulas and their tableaux; and third, it demonstrates the flexibility of relational provers to mimic the full spectrum of theorem provers, from fully automated to fully dependent on the user.

We proceed as follows. In section 2 we provide a brief description of αKanren and describe the concept of tableau theorem proving. In section 3 we motivate our declarative prover by examining its declarative properties and the proofs it returns. In section 4 we present the implementation of αlean*TAP*. In section 5 we briefly examine αlean*TAP*'s performance. In section 6, we discuss related work. Familiarity with αKanren and knowledge of tableau theorem proving would be helpful; for more on these topics, see the references given in section 2.

2 Preliminaries

We begin by presenting a brief overview of αKanren, the language in which αlean*TAP* is written. We also provide an introduction to tableau theorem proving and its implementation in lean*TAP*.

2.1 αKanren Refresher

αKanren is an embedding of nominal logic programming in Scheme. It extends the Scheme language with a term constructor $\bowtie$ (pronounced "tie") and five operators: $\equiv$, $\#$, **exist**[1], **fresh**, and **cond**^{e}. In addition to these declarative operators, we use the impure operator **cond**^{a} to model Prolog's cut.

$\equiv$ unifies two terms using nominal unification. **exist** and **fresh**, which are syntactically similar to Scheme's **lambda** and whose bodies are conjoined, are used to introduce new lexical variables; those introduced by **exist** bind logic (or unification) variables, while those introduced by **fresh** bind *noms* (also called

[1] The name **exist** is chosen to avoid conflict with R^{6}RS Scheme's [6] *exists*.

"names" or "atoms" in nominal logic). A nom unifies only with a logic variable or with itself; in αlean*TAP*, noms represent variable names. # is a freshness constraint: $(\# \ a \ t)$ asserts that the nom a does *not* occur free in t. $\bowtie$ is a term constructor: $(\bowtie \ a \ t)$ creates a term in which all free occurrences of the nom a in t are considered bound. Thus $(\# \ a \ (\bowtie \ a \ t))$ always succeeds.

cond^e, which is syntactically similar to **cond**, expresses a disjunction of clauses. Each clause may contain arbitrarily many conjoined goals. **cond**^{a} is similar to **cond**^{e}, but only a single clause of a **cond**^{a} may succeed. The successful clause may succeed an arbitrary number of times, but once its first goal is successful, no other clause may succeed. This behavior is similar to placing a cut (!) before the first conjunct in the body of each relevant clause.

run provides an interface between Scheme and αKanren; it allows the user to limit the number of answers returned, and to specify a logic variable whose value should be *reified* to obtain answers. Reification is the process of replacing distinct logic variables in a term with unique names. The first such variable to be found is represented by the symbol $\_{0}$, the second by $\_{1}$, and so on. For example:

```
(run⁵ (q)
  (exist (x y z)
    (condᵉ
      ((≡ x 3) (≡ y 2) (≡ z y))          ⇒ ((3 2 2) (_₀ _₀ _₀) (_₀ _₁ _₀))
      ((≡ x y) (≡ y z))
      ((≡ x z)))
    (≡ `(,x ,y ,z)² q)))
```

This **run** expression has three answers, each corresponding to one line of the **cond**^{e}. In the first answer, all three variables have been instantiated to ground values. In the second, the three variables have been unified with one another, so they have the same reified value. In the third, x and z share the same reified value, which is distinct from that of y.

Nominal unification equates α-equivalent binders:

```
(run¹ (q) (fresh (a b) (≡ (⋈ a a) (⋈ b b)))) ⇒ (_₀)
```

Although the noms a and b are distinct and would therefore fail to unify, this **run** expression succeeds. Like the terms $\lambda a.a$ and $\lambda b.b$, the terms $(\bowtie \ a \ a)$ and $(\bowtie \ b \ b)$ bind in the same way and are thus α-equivalent.

For a more complete description of αKanren, see Byrd and Friedman [2]. A newer implementation of αKanren in R^{6}RS Scheme [6] was used in the development of αlean*TAP*[3]; this version uses triangular substitutions [7] instead of idempotent substitutions and is significantly faster. αKanren is based on αProlog [8], which implements the nominal unification of Urban, Pitts, and Gabbay [5], and miniKanren, an earlier logic programming language [9, 10].

[2] Here, backquote and comma are used to build a list of logic variables: the expression `(,x ,y ,z) is equivalent to `[X, Y, Z]` in Prolog. Similarly, the expression `(,x . ,y) constructs a pair, and is equivalent to `[X|Y]` in Prolog.

[3] The latest αKanren and αlean*TAP* source code is available at `https://code.launchpad.net/~jnear-csail/minikanren/alphaleanTAP`.

2.2 Tableau Theorem Proving

Tableau is a method of proving first-order theorems that works by refuting the theorem's negation. In our description we assume basic knowledge of first-order logic; for coverage of this subject and a more complete description of tableau proving, see Fitting [11]. For simplicity, we consider only formulas in Skolemized *negation normal form* (NNF). Converting a formula to this form requires removing existential quantifiers through Skolemization, reducing logical connectives so that only $\wedge$, $\vee$, and $\neg$ remain, and pushing negations inward until they are applied only to literals—see section 3 of Beckert and Posegga [1] for details.

To form a tableau, a compound formula is expanded into branches recursively until no compound formulas remain. The leaves of this tree structure are referred to as *literals*. lean*TAP* forms and expands the tableau according to the following rules. When the prover encounters a conjunction $x \wedge y$, it expands both x and y on the same branch. When the prover encounters a disjunction $x \vee y$, it splits the tableau and expands x and y on separate branches. Once a formula has been fully expanded into a tableau, it can be proved unsatisfiable if on each branch of the tableau there exist two complementary literals a and $\neg a$ (each branch is *closed*). In the case of propositional logic, syntactic comparison is sufficient to find complementary literals; in first-order logic, sound unification must be used. A closed tableau represents a proof that the original formula is unsatisfiable.

The addition of universal quantifiers makes the expansion process more complicated. To prove a universally quantified formula $\forall x.M$, lean*TAP* generates a logic variable v and expands M, replacing all occurrences of x with v (i.e., it expands M' where $M' = M[v/x]$). If lean*TAP* is unable to close the current branch after this expansion, it has the option of generating another logic variable and expanding the original formula again. When the prover expands the formula $\forall x.F(x) \wedge (\neg F(\mathsf{a}) \vee \neg F(\mathsf{b}))$, for example, $\forall x.F(x)$ must be expanded twice, since x cannot be instantiated to both a and b.

3 Introducing αlean*TAP*

We begin by presenting some examples of αlean*TAP*'s abilities, both in proving ground theorems and in generating theorems. We also explore the proofs generated by αlean*TAP*, and show how passing partially-instantiated proofs to the prover can greatly improve its performance.

3.1 Running Forwards

Both lean*TAP* and αlean*TAP* can prove ground theorems; in addition, αlean*TAP* produces a proof. This proof is a list representing the steps taken to build a closed tableau for the theorem; Paulson [12] has shown that translation to a more standard format is possible. Since a closed tableau represents an unsatisfiable formula, such a list of steps proves that the negation of the formula is valid. If the list of steps is ground, the proof search becomes deterministic, and αlean*TAP* acts as a proof checker.

lean*TAP* encodes first-order formulas using Prolog terms. For example, the term `(p(b),all(X,(-p(X);p(s(X)))))` represents $p(\mathsf{b}) \wedge \forall x.\neg p(x) \vee p(s(x))$. In our prover, we represent formulas using Scheme lists with extra tags:

```
(and (pos (app p (app b))) (forall (⋈ a (or (neg (app p (var a)))
                                              (pos (app p (app s (var a))))))))
```

Consider Pelletier Problem 18 [13]: $\exists y.\forall x.F(y) \Rightarrow F(x)$. To prove this theorem in αlean*TAP*, we transform it into the following *negation* of the NNF:

```
(forall (⋈ a (and (pos (app f (var a))) (neg (app f (app g1 (var a)))))))
```

where (app g1 (var a)) represents the application of a Skolem function to the universally quantified variable a. Passing this formula to the prover, we obtain the proof (univ conj savefml savefml univ conj close). This proof lists the steps the prover (presented in section 4.3) follows to close the tableau. Because both conjuncts of the formula contain the nom a, we must expand the universally quantified formula more than once.

Partially instantiating the proof helps αlean*TAP* prove theorems with similar subparts. We can create a non-ground proof that describes in general how to prove the subparts and have αlean*TAP* fill in the trivial differences. This can speed up the search for a proof considerably. By inspecting the negated NNF of Pelletier Problem 21, for example, we can see that there are at least two portions of the theorem that will have the same proof. By specifying the structure of the first part of the proof and constraining the identical portions by using the same logic variable to represent both, we can give the prover some guidance without specifying the whole proof. We pass the following non-ground proof to αlean*TAP*:

```
(conj univ split (conj savefml savefml conj split x x)
      (conj savefml savefml conj split (close) (savefml split y y)))
```

On our test machine, our prover solves the original problem with no help in 68 milliseconds (ms); given the knowledge that the later parts of the proof will be duplicated, the prover takes only 27 ms. This technique also yields improvement when applied to Pelletier Problem 43: inspecting the negated NNF of the formula, we see two parts that look nearly identical. The first part of the negated NNF—the part representing the theorem itself—has the following form:

```
(and (or (and (neg (app Q (app g4) (app g3)))
              (pos (app Q (app g3) (app g4))))
         (and (pos (app Q (app g4) (app g3)))
              (neg (app Q (app g3) (app g4))))) ...)
```

Since we suspect that the same proof might suffice for both branches of the theorem, we give the prover the partially-instantiated proof (conj split x x). Given just this small amount of help, αlean*TAP* proves the theorem in 720 ms, compared to 1.5 seconds when the prover has no help at all. While situations in which large parts of a proof are identical are rare, this technique also allows us to handle situations in which different parts of a proof are merely similar by instantiating as much or as little of the proof as necessary.

3.2 Running Backwards

Unlike lean*TAP*, αlean*TAP* can generate valid theorems. Some interpretation of the results is required since the theorems generated are negated formulas in NNF.[4] In the example

```
(run¹ (q) (exist (x) (proveᵒ q '() '() '() x)))
 ⇒ ((and (pos (app _₀)) (neg (app _₀))))
```

the reified logic variable $_{-0}$ represents any first-order formula p, and the entire answer represents the formula $p \wedge \neg p$. Negating this formula yields the original theorem: $\neg p \vee p$, or the law of excluded middle. We can also generate more complicated theorems; here we use the "generate and test" idiom to find the first theorem matching the negated NNF of the inference rule *modus ponens*:

```
(run¹ (q)
  (exist (x)
    (proveᵒ x '() '() '() q)
    (≡ '(and (and (or (neg (app a)) (pos (app b))) (pos (app a)))
             (neg (app b)))
         x)))
⇒ ((conj conj split (savefml close) (savefml savefml close)))
```

This process takes about 5.1 seconds; *modus ponens* is the 173rd theorem to be generated, and the prover also generates a proof of its validity. When this proof is given to αlean*TAP*, *modus ponens* is the sixth theorem generated, and the process takes only 20 ms.

Thus the declarative nature of αlean*TAP* is useful both for generating theorems and for producing proofs. Due to this flexibility, αlean*TAP* could become the core of a larger proof system. Automated theorem provers like lean*TAP* are limited in the complexity of the problems they can solve, but given the ability to accept assistance from the user, more problems become tractable.

As an example, consider Pelletier Problem 47: Schubert's Steamroller. This problem is difficult for tableau-based provers like lean*TAP* and αlean*TAP*, and neither can solve it automatically [1]. Given some help, however, αlean*TAP* can prove the Steamroller. Our approach is to prove a series of smaller lemmas that act as stepping stones toward the final theorem; as each lemma is proved, it is added as an assumption in proving the remaining ones. The proof process is automated—the user need only specify which lemmas to prove and in what order. Using this strategy, αlean*TAP* proves the Steamroller in about five seconds; the proof requires twenty lemmas.

αlean*TAP* thus offers an interesting compromise between large proof assistants and smaller automated provers. It achieves some of the capabilities of a larger system while maintaining the lean deduction philosophy introduced by lean*TAP*. Like an automated prover, it is capable of proving simple theorems without user guidance. Confronted with a more complex theorem, however, the user

[4] The full implementation of αlean*TAP* includes a simple declarative translator from negated NNF to a positive form.

can provide a partially-instantiated proof; αlean*TAP* can then check the proof and fill in the trivial parts the user has left out. Because αlean*TAP* is declarative, the user may even leave required axioms out of the theorem to be proved and have the system derive them. This flexibility comes at no extra cost to the user—the prover remains both concise and reasonably efficient.

The flexibility of αlean*TAP* means that it could be made interactive through the addition of a read-eval-print loop and a simple proof translator between αlean*TAP*'s proofs and a more human-readable format. Since the proof given to αlean*TAP* may be partially instantiated, such an interface would allow the user to conveniently guide αlean*TAP* in proving complex problems. With the addition of equality and the ability to perform single beta steps, this flexibility would become more interesting—in addition to reasoning about programs and proving properties about them, αlean*TAP* would instantiate non-ground programs during the proof process.

4 Implementation

We now present the implementation of αlean*TAP*. We begin with a translation of lean*TAP* from Prolog into αKanren. We then show how to eliminate the translation's impure features through a combination of substitution and tagging.

lean*TAP* implements both expansion and closing of the tableau. When the prover encounters a conjunction, it uses its argument `UnExp` as a stack (Figure 1): lean*TAP* expands the first conjunct, pushing the second onto the stack for later expansion. If the first conjunct cannot be refuted, the second is popped off the stack and expansion begins again. When a disjunction is encountered, the split in the tableau is reflected by two recursive calls. When a universal quantifier is encountered, the quantified variable is replaced by a new logic variable, and the formula is expanded. The `FreeV` argument is used to avoid replacing the free variables of the formula. lean*TAP* keeps a list of the literals it has encountered on the current branch of the tableau in the argument `Lits`. When a literal is encountered, lean*TAP* attempts to unify its negation with each literal in `Lits`; if any unification succeeds, the branch is closed. Otherwise, the current literal is added to `Lits` and expansion continues with a formula from `UnExp`.

4.1 Translation to αKanren

While αKanren is similar to Prolog with the addition of nominal unification, αKanren also uses a variant of interleaving depth-first search [14], so the order of **cond**^{e} clauses in αKanren is irrelevant. Because of Prolog's depth-first search, lean*TAP* must use `VarLim` to limit its search depth; in αKanren, `VarLim` is not necessary, and thus we omit it.

In Figure 1 we present mKlean*TAP*, our translation of lean*TAP* into αKanren; we label two clauses (①, ②), since we will modify these clauses later. To express Prolog's cuts, our definition uses **cond**^{a}. The final two clauses of lean*TAP* do not contain Prolog cuts; in mKlean*TAP*, they are combined into a single clause containing a **cond**^{e}. In place of lean*TAP*'s recursive call to `prove` to check the membership of `Lit` in `Lits`, we call *member*^{o}, which performs a membership check using sound unification.[5] Prolog's `copy_term/2` is not built into αKanren; this addition is available as part of the mKlean*TAP* source code.

```
prove((E1,E2),UnExp,Lits,
      FreeV,VarLim) :- !,
  prove(E1,[E2|UnExp],Lits,
        FreeV,VarLim).

prove((E1;E2),UnExp,Lits,
      FreeV,VarLim) :- !,
  prove(E1,UnExp,Lits,FreeV,VarLim),
  prove(E2,UnExp,Lits,FreeV,Varlim).

prove(all(X,Fml),UnExp,Lits,
      FreeV,VarLim) :- !,
  \+ length(FreeV,VarLim),
  copy_term((X,Fml,FreeV),
            (X1,Fml1,FreeV)),
  append(UnExp,[all(X,Fml)],UnExp1),
  prove(Fml1,UnExp1,Lits,
        [X1|FreeV],VarLim).

prove(Lit,_,[L|Lits],_,_) :-
  (Lit = -Neg; -Lit = Neg) ->
   (unify(Neg,L);
    prove(Lit,[],Lits,_,_)).

prove(Lit,[Next|UnExp],Lits,
      FreeV,VarLim) :-
  prove(Next,UnExp,[Lit|Lits],
        FreeV,VarLim).
```

```
(define prove^o
  (λ (fml unexp lits freev)
    (cond^a
      ((exist (e1 e2)
         (≡ `(and ,e1 ,e2) fml)
         (prove^o e1 `(,e2 . ,unexp) lits freev)))
      ((exist (e1 e2)
         (≡ `(or ,e1 ,e2) fml)
         (prove^o e1 unexp lits freev)
         (prove^o e2 unexp lits freev)))
    ① ((exist (x x1 body body1 unexp1)
         (≡ `(forall ,x ,body) fml)
         (copy-term^o `(,x ,body ,freev)
                      `(,x1 ,body1 ,freev))
         (append^o unexp `(,fml) unexp1)
         (prove^o body1 unexp1 lits
                  `(,x1 . ,freev))))
    ② ((cond^e
         ((exist (neg)
            (cond^a
              ((≡ `(not ,neg) fml))
              ((≡ `(not ,fml) neg)))
            (member^o neg lits)))
         ((exist (next unexp1)
            (≡ `(,next . ,unexp1) unexp)
            (prove^o next unexp1 `(,fml . ,lits)
                     freev))))))))
```

Fig. 1. lean*TAP* and mKlean*TAP*: a translation from Prolog to αKanren

4.2 Eliminating *copy-term*^{o}

Since *copy-term*^{o} is an impure operator, its use makes *prove*^{o} non-declarative: reordering the goals in the prover can result in different behavior. For example, moving the call to *copy-term*^{o} after the call to *prove*^{o} causes the prover to diverge when given any universally quantified formula. To make our prover declarative, we must eliminate the use of *copy-term*^{o}.

Tagging the logic variables that represent universally quantified variables allows the use of a declarative technique that creates two pristine copies of the

[5] We define *member*^{o} in Figure 3; it uses sound unification ($\equiv^{\checkmark}$).

original term: one copy may be expanded and the other saved for later copying. Unfortunately, this copying examines the entire body of each quantified formula and instantiates the original term to a potentially invalid formula.

Another approach is to represent quantified variables with symbols or strings. When a new instantiation is needed, a new variable name can be generated, and the new name can be substituted for the old without affecting the original formula. This solution does not destroy the prover's input, but it is difficult to ensure that the provided data is in the correct form declaratively: if the formula to be proved is non-ground, then the prover must generate unique names. If the formula *does* contain these names, however, the prover must *not* generate new ones. This problem can be solved with a declarative preprocessor that expects a logical formula *without* names and puts them in place. If the preprocessor is passed a non-ground formula, it instantiates the formula to the correct form. The requirement of a preprocessor, however, means the prover itself is not declarative.

We use nominal logic [4] to solve the *copy-term*^{o} problem. Nominal logic is designed to handle the complexities of dealing with names and binders declaratively. Since noms represent unique names, we achieve the benefits of the symbol or string approach without the use of a preprocessor. We can generate unique names each time we encounter a universally quantified formula, and use nominal unification to perform the renaming of the quantified variable. If the original formula is uninstantiated, our newly-generated name is unique and is put in place correctly; we no longer need a preprocessor to perform this function.

Using the tools of nominal logic, we can modify mKlean*TAP* to represent universally quantified variables using noms and to perform substitution instead of copying. When the prover reaches a literal, however, it must replace each nom with a logic variable, so that unification may successfully compare literals. To accomplish this, we associate a logic variable with each unique nom, and replace every nom with its associated variable before comparing literals. These variables are generated each time the prover expands a quantified formula.

To implement this strategy, we change our representation of formulas slightly. Instead of representing $\forall x.F(x)$ as (forall x (f x)), we use a nom wrapped in a var tag to represent a variable reference, and the term constructor $\bowtie$ to represent the $\forall$ binder: (forall ($\bowtie$ a (f (var a)))), where a is a nom. The var tag allows us to distinguish noms representing variables from other formulas. We now write a relation *subst-lit*^{o} to perform substitution of logic variables for tagged noms in a literal, and we modify the literal case of *prove*^{o} to use it. We also replace the clause handling forall formulas and define *lookup*^{o}. The two clauses of *lookup*^{o} overlap, but since each mapping in the environment is from a unique nom to a logic variable, a particular nom will never appear twice.

We present the changes needed to eliminate *copy-term*^{o} from mKlean*TAP* in Figure 2. Instead of copying the body of each universally quantified formula, we generate a logic variable x and add an association between the nom representing the quantified variable and x to the current environment. When we prepare to close a branch of the tableau, we call *subst-lit*^{o}, replacing the noms in the current literal with their associated logic variables.

```
① ((fresh (a)
      (exist (x body unexp₁)
        (≡ `(forall ,(⋈ a body)) fml)
        (appendᵒ unexp `(,fml) unexp₁)
        (proveᵒ body unexp₁ lits
                `((,a . ,x) . ,env)))))

② ((exist (lit)
      (subst-litᵒ fml env lit)
      (condᵉ
        ((exist (neg)
           (condᵃ
             ((≡ `(not ,neg) lit))
             ((≡ `(not ,lit) neg)))
           (memberᵒ neg lits)))
        ((exist (next unexp₁)
           (≡ `(,next . ,unexp₁) unexp)
           (proveᵒ next unexp₁ `(,lit . ,lits)
                   env))))))
```

```
(define lookupᵒ
  (λ (a env out)
    (exist (first rest)
      (condᵉ
        ((≡ `((,a . ,out) . ,rest) env))
        ((≡ `(,first . ,rest) env)
         (lookupᵒ a rest out))))))

(define subst-litᵒ
  (λ (fml env out)
    (condᵃ
      ((exist (a)
         (≡ `(var ,a) fml)
         (lookupᵒ a env out)))
      ((exist (e₁ e₂ r₁ r₂)
         (≡ `(,e₁ . ,e₂) fml)
         (≡ `(,r₁ . ,r₂) out)
         (subst-litᵒ e₁ env r₁)
         (subst-litᵒ e₂ env r₂)))
      ((≡ fml out)))))
```

Fig. 2. Changes to mKleanTAP to eliminate *copy-term*^{o}

The original `copy_term/2` approach used by leanTAP and mKleanTAP avoids replacing free variables by copying the list (*x body freev*). The copied version is unified with the list (x_1 $body_1$ *freev*), so that *only* the variable x will be replaced by a new logic variable—the free variables will be copied, but those copies will be unified with the original variables afterwards. Since our substitution strategy does not affect free variables, the *freev* argument is no longer needed, and so we have eliminated it.

4.3 Eliminating cond^a

Both *prove*^{o} and *subst-lit*^{o} use **cond**^{a} because the clauses that recognize literals overlap with the other clauses. To solve this problem, we have designed a tagging scheme that ensures that the clauses of our substitution and *prove*^{o} relations do not overlap. To this end, we tag both positive and negative literals, applications, and variables. Constants are represented by applications of zero arguments. Our prover thus accepts formulas of the following form:

$$
\begin{array}{lcl}
\mathit{Fml} & \rightarrow & (\mathsf{and}\ \mathit{Fml}\ \mathit{Fml}) \mid (\mathsf{or}\ \mathit{Fml}\ \mathit{Fml}) \mid (\mathsf{forall}\ (\bowtie\ \mathit{Nom}\ \mathit{Fml})) \mid \mathit{Lit} \\
\mathit{Lit} & \rightarrow & (\mathsf{pos}\ \mathit{Term}) \mid (\mathsf{neg}\ \mathit{Term}) \\
\mathit{Term} & \rightarrow & (\mathsf{var}\ \mathit{Nom}) \mid (\mathsf{app}\ \mathit{Symbol}\ \mathit{Term}^*)
\end{array}
$$

This scheme has been chosen carefully to allow unification to compare literals. In particular, the tags on variables *must* be discarded before literals are compared. Consider the two non-ground literals (not (f x)) and (f (p y)). These literals are complementary: the negation of one unifies with the other, associating x with (p y). When we apply our tagging scheme, however, these literals become (neg (app f (var x))) and (pos (app f (app p (var y)))), respectively, and are no longer complementary: their subexpressions (var x) and (app p (var y)) do not unify. To avoid this problem, our substitution relation discards the var tag when it replaces noms with logic variables.

```
(define prove^o
  (λ (fml unexp lits env proof)
    (cond^e
      ((exist (e1 e2 prf)
         (≡ `(and ,e1 ,e2) fml)
         (≡ `(conj . ,prf) proof)
         (prove^o e1 `(,e2 . ,unexp)
                  lits env prf)))
      ((exist (e1 e2 prf1 prf2)
         (≡ `(or ,e1 ,e2) fml)
         (≡ `(split ,prf1 ,prf2) proof)
         (prove^o e1 unexp lits env prf1)
         (prove^o e2 unexp lits env prf2)))
      ((fresh (a)
         (exist (x body unexp1 prf)
           (≡ `(forall ,(⋈ a body)) fml)
           (≡ `(univ . ,prf) proof)
           (append^o unexp `(,fml) unexp1)
           (prove^o body unexp1 lits
                    `((,a . ,x) . ,env) prf))))
      ((exist (lit)
         (subst-lit^o fml env lit)
         (cond^e
           ((exist (tm neg)
              (≡ `(close) proof)
              (cond^e
                ((≡ `(pos ,tm) lit)
                 (≡ `(neg ,tm) neg))
                ((≡ `(neg ,tm) lit)
                 (≡ `(pos ,tm) neg)))
              (member^o neg lits)))
           ((exist (next unexp1 prf)
              (≡ `(,next . ,unexp1) unexp)
              (≡ `(savefml . ,prf) proof)
              (prove^o next unexp1 `(,lit . ,lits)
                       env prf)))))))))

(define member^o
  (λ (x ls)
    (exist (a d)
      (≡ `(,a . ,d) ls)
      (cond^e
        ((≡^√ a x))
        ((member^o x d))))))
```

```
(define append^o
  (λ (ls s out)
    (cond^e
      ((≡ '() ls) (≡ s out))
      ((exist (a d r)
         (≡ `(,a . ,d) ls)
         (≡ `(,a . ,r) out)
         (append^o d s r))))))

(define subst-lit^o
  (λ (fml env out)
    (cond^e
      ((exist (l r)
         (≡ `(pos ,l) fml)
         (≡ `(pos ,r) out)
         (subst-term^o l env r)))
      ((exist (l r)
         (≡ `(neg ,l) fml)
         (≡ `(neg ,r) out)
         (subst-term^o l env r))))))

(define subst-term^o
  (λ (fml env out)
    (cond^e
      ((exist (a)
         (≡ `(var ,a) fml)
         (lookup^o a env out)))
      ((exist (f d r)
         (≡ `(app ,f . ,d) fml)
         (≡ `(app ,f . ,r) out)
         (subst-term*^o d env r))))))

(define subst-term*^o
  (λ (tm* env out)
    (cond^e
      ((≡ '() tm*) (≡ '() out))
      ((exist (e1 e2 r1 r2)
         (≡ `(,e1 . ,e2) tm*)
         (≡ `(,r1 . ,r2) out)
         (subst-term^o e1 env r1)
         (subst-term*^o e2 env r2))))))
```

Fig. 3. Final definition of αlean*TAP*

Given our new tagging scheme, we can easily rewrite our substitution relation without the use of **cond**^{a}. We simply follow the production rules of the grammar, defining a relation to recognize each.

Finally, we modify *prove*^{o} to take advantage of the same tags. We also add a *proof* argument to *prove*^{o}. We call this version of the prover αlean*TAP*, and present its definition in Figure 3. It is declarative, since we have eliminated the use of *copy-term*^{o} and every use of **cond**^{a}. In addition to being a sound and complete theorem prover for first-order logic, αlean*TAP* can now generate valid first-order theorems.

5 Performance

Like the original lean*TAP*, αlean*TAP* can prove many theorems in first-order logic. Because it is declarative, αlean*TAP* is generally slower at proving ground theorems than mKlean*TAP*, which is slower than the original lean*TAP*. Figure 4 presents a summary of αlean*TAP*'s performance on the first 46 of Pelletier's 75 problems [13], showing it to be roughly twice as slow as mKlean*TAP*.

#	lean*TAP*	mKlean*TAP*	αlean*TAP*
1	0.1	0.7	2.0
2	0.0	0.1	0.3
3	0.0	0.2	0.5
4	0.0	1.0	1.7
5	0.1	1.2	2.5
6	0.0	0.1	0.2
7	0.0	0.1	0.2
8	0.0	0.3	0.8
9	0.1	4.3	9.7
10	0.3	5.5	10.2
11	0.0	0.3	0.6
12	0.6	17.7	31.9
13	0.1	3.7	8.2
14	0.1	4.2	9.7
15	0.0	0.8	1.9
16	0.0	0.2	0.6
17	1.1	9.2	18.1
18	0.1	0.5	1.2
19	0.3	15.1	33.5
20	0.5	8.1	12.7
21	0.4	22.1	38.7
22	0.1	3.4	6.4
23	0.1	2.5	5.4

#	lean*TAP*	mKlean*TAP*	αlean*TAP*
24	1.7	31.9	60.3
25	0.2	7.5	14.1
26	0.8	130.9	187.5
27	2.3	40.4	79.3
28	0.3	19.1	29.6
29	0.1	27.9	57.0
30	0.1	4.2	9.6
31	0.3	13.2	23.1
32	0.2	23.9	42.4
33	0.1	15.9	39.2
34	199129.0	7272.9	8493.5
35	0.1	0.5	1.1
36	0.2	6.7	12.4
37	0.8	123.3	169.2
38	8.9	4228.8	8363.8
39	0.0	1.1	2.8
40	0.2	8.1	19.2
41	0.1	6.9	17.0
42	0.4	15.0	32.1
43	43.2	668.4	1509.6
44	0.3	15.1	35.7
45	3.4	145.3	239.7
46	7.7	505.5	931.2

Fig. 4. Performance of lean*TAP*, mKlean*TAP*, and αlean*TAP* on the first 46 Pelletier Problems. All times are in milliseconds, averaged over 100 trials. All tests were run under Debian Linux on an IBM Thinkpad X40 with a 1.1GHz Intel Pentium-M processor and 768MB RAM. lean*TAP* tests were run under SWI-Prolog 5.6.55; mKlean*TAP* and αlean*TAP* tests were run under Ikarus Scheme 0.0.3+.

These performance numbers suggest that while there is a penalty to be paid for declarativeness, it is not so severe as to cripple the prover. The advantage mKlean*TAP* enjoys over the original lean*TAP* in Problem 34 is due to αKanren's interleaving search strategy; as the result for mKlean*TAP* shows, the original lean*TAP* is faster than αlean*TAP* for any given search strategy.

Many automated provers now use the TPTP problem library [15] to assess performance. Even though it is faster than αlean*TAP*, however, lean*TAP* solves few of the TPTP problems. The Pelletier Problems, on the other hand, fall into the class of theorems lean*TAP* was designed to prove, and so we feel they provide a better set of tests for the comparison between lean*TAP* and αlean*TAP*.

6 Related Work

Through his integration of lean*TAP* with the Isabelle theorem prover [12], Paulson shows that it is possible to modify lean*TAP* to produce a list of Isabelle tactics representing a proof. This approach could be reversed to produce a proof translator from Isabelle proofs to αlean*TAP* proofs, allowing αlean*TAP* to become interactive as discussed in section 3.2.

The lean*TAP* Frequently Asked Questions [16] states that lean*TAP* might be made declarative through the elimination of Prolog's cuts but does not address the problem of `copy_term/2` or specify how the cuts might be eliminated. Other provers written in Prolog include those of Manthey and Bry [17] and Stickel [18], but each uses some impure feature and is thus not declarative.

Christiansen [19] uses constraint logic programming and metavariables (similar to nominal logic's names) to build a declarative interpreter based on Kowalski's non-declarative `demonstrate` predicate [20]. This approach is similar to ours, but the Prolog-like language is not complicated by the presence of binders.

Higher-order abstract syntax (HOAS), presented in Pfenning and Elliott [21], can be used instead of nominal logic to perform substitution on quantified formulas. Felty and Miller [22] were among the first to develop a theorem prover using HOAS to represent formulas; Pfenning and Schurmann [23] also use a HOAS encoding for formulas.

Kiselyov [24] uses a HOAS encoding for universally quantified formulas in his original translation of lean*TAP* into miniKanren. Since miniKanren does not implement higher-order unification, the prover cannot generate theorems.

Lisitsa's λlean*TAP* [25] is a prover written in λProlog that addresses the problem of `copy_term/2` using HOAS, and is perhaps closest to our own work. Like αlean*TAP*, λlean*TAP* replaces universally quantified variables with logic variables using substitution. However, λlean*TAP* is not declarative, since it contains cuts. Even if we use our techniques to remove the cuts from λlean*TAP*, the prover does not generate theorems, since λProlog uses a depth-first search strategy. Generating theorems requires the addition of a tagging scheme and iterative deepening on *every clause* of the program. Even with these additions, however, λlean*TAP* often generates theorems that do not have the proper HOAS encoding, since that encoding is not specified in the prover.

7 Conclusion

We have presented αleanTAP, a declarative tableau theorem prover for first-order classical logic. Based on the concise but non-declarative prover leanTAP, αleanTAP retains leanTAP's minimalism without the use of Prolog's `copy_term/2` or cut. To avoid the use of `copy_term/2`, we have represented universally quantified variables with noms rather than logic variables, allowing us to perform substitution instead of copying. To eliminate cuts, we have enhanced the tagging scheme for representing formulas.

Both of these transformations are broadly applicable. When cuts are used to handle overlapping clauses, a carefully crafted tagging scheme can often be used to eliminate overlapping. When terms must be copied, substitution can often be used instead of `copy_term/2`—in the case of αleanTAP, we use a combination of nominal unification and substitution.

The resulting theorem prover retains the strengths of leanTAP. It is slower than mKleanTAP, our translation of leanTAP, by a factor of two, but remains concise. In addition, its declarative nature makes it more flexible than leanTAP: given non-ground values for both the theorem to be proved *and* its proof, αleanTAP fills in the uninstantiated parts. Like leanTAP, αleanTAP has the capability of proving theorems on its own, and like a proof assistant, it can accept help from the user in proving theorems.

Acknowledgements

We thank Oleg Kiselyov for pointing out an alternative solution for making leanTAP declarative and for his helpful comments on a draft of this paper. We also thank Matthew Lakin for his comments on a later version. We are grateful to Ramana Kumar and Christian Urban for their work on the triangular substitution-based implementation of αKanren. We also thank Micah Linnemeier and Adam Hinz for their participation in the early stages of this research. We appreciate the many insightful comments provided by the anonymous referees.

References

1. Beckert, B., Posegga, J.: leanTAP: Lean tableau-based deduction. Journal of Automated Reasoning **15**(3) (1995) 339–358
2. Byrd, W.E., Friedman, D.P.: αKanren: A fresh name in nominal logic programming. Proceedings of the 2007 Workshop on Scheme and Functional Programming, Université Laval Technical Report DIUL-RT-0701 79–90 (*see also* `http://www.cs.indiana.edu/~webyrd` *for improvements*)
3. Mellish, C.S.: The Automatic Generation of Mode Declarations for Prolog Programs. Dept. of Artificial Intelligence, University of Edinburgh (1981)
4. Pitts, A.M.: Nominal logic: A first order theory of names and binding. Lecture Notes in Computer Science **2215** (2001) 219–242
5. Urban, C., Pitts, A., Gabbay, M.: Nominal unification. Theoretical Computer Science **323**(1-3) (2004) 473–497

6. Sperber, M., Clinger, W., Dybvig, R., Flatt, M., van Straaten, A., Kelsey, R., Rees, J.: Revised 6 report on the algorithmic language Scheme (September 2007)
7. Baader, F., Snyder, W.: Unification theory. Handbook of Automated Reasoning **1** 446–533
8. Cheney, J., Urban, C.: αProlog: A logic programming language with names, binding and α-equivalence. Lecture Notes in Computer Science **3132** (2004) 269–283
9. Byrd, W.E., Friedman, D.P.: From variadic functions to variadic relations
10. Friedman, D.P., Byrd, W.E., Kiselyov, O.: The Reasoned Schemer. The MIT Press (2005)
11. Fitting, M.: First-Order Logic and Automated Theorem Proving. Springer (1996)
12. Paulson, L.C.: A generic tableau prover and its integration with Isabelle. Journal of Universal Computer Science **5**(3) (1999) 73–87
13. Pelletier, F.: Seventy-five problems for testing automatic theorem provers. Journal of Automated Reasoning **2**(2) (1986) 191–216
14. Kiselyov, O., Shan, C., Friedman, D., Sabry, A.: Backtracking, interleaving, and terminating monad transformers: (functional pearl). ACM SIGPLAN Notices **40**(9) (2005) 192–203
15. Sutcliffe, G., Suttner, C.: The TPTP Problem Library. Journal of Automated Reasoning **21**(2) (1998) 135–277
16. Beckert, B., Posegga, J.: The lean*TAP*-FAQ: Frequently asked questions about lean*TAP*. `http://www.uni-koblenz.de/~beckert/pub/LeanTAP_FAQ.pdf`
17. Manthey, R., Bry, F.: SATCHMO: A theorem prover implemented in Prolog. Proceedings of the 9th International Conference on Automated Deduction (1988) 415–434
18. Stickel, M.: A Prolog technology theorem prover. Proceedings of the 9th International Conference on Automated Deduction (1988) 752–753
19. Christiansen, H.: Automated reasoning with a constraint-based metainterpreter. The Journal of Logic Programming **37**(1-3) (1998) 213–254
20. Kowalski, R.A.: Logic for Problem Solving. Prentice Hall PTR, Upper Saddle River, NJ, USA (1979)
21. Pfenning, F., Elliot, C.: Higher-order abstract syntax. Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation **23**(7) (1988) 199–208
22. Felty, A., Miller, D.: Specifying theorem provers in a higher-order logic programming language. Proceedings of the 9th International Conference on Automated Deduction (1988) 61–80
23. Pfenning, F., Schurmann, C.: System description: Twelf—a meta-logical framework for deductive systems. Proceedings of the 16th International Conference on Automated Deduction (1999) 202–206
24. Friedman, D.P., Kiselyov, O.: A declarative applicative logic programming system. `http://kanren.sourceforge.net`
25. Lisitsa, A.: λleanTAP: lean deduction in λProlog. Technical report, ULCS-03-017, University of Liverpool, Department of Computer Science, 2003