J Autom Reasoning (2013) 51:57–77
DOI 10.1007/s10817-013-9283-8

Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems

Chad E. Brown

Received: 6 February 2013 / Accepted: 18 February 2013 / Published online: 15 March 2013

Abstract We describe a complete theorem proving procedure for higher-order logic that uses SAT-solving to do much of the heavy lifting. The theoretical basis for the procedure is a complete, cut-free, ground refutation calculus that incorporates a restriction on instantiations. The refined nature of the calculus makes it conceivable that one can search in the ground calculus itself, obtaining a complete procedure without resorting to meta-variables and a higher-order lifting lemma. Once one commits to searching in a ground calculus, a natural next step is to consider ground formulas as propositional literals and the rules of the calculus as propositional clauses relating the literals. With this view in mind, we describe a theorem proving procedure that primarily generates relevant formulas along with their corresponding propositional clauses. The procedure terminates when the set of propositional clauses is unsatisfiable. We prove soundness and completeness of the procedure. The procedure has been implemented in a new higher-order theorem prover, Satallax, which makes use of the SAT-solver MiniSat. We also describe the implementation and give several examples. Finally, we include experimental results of Satallax on the higher-order part of the TPTP library.

Keywords Higher-order logic • Simple type theory • Higher-order theorem proving • Abstract consistency • SAT solving

1 Introduction

There are a number of distinct aspects of automated theorem proving. First, there is the usual combinatorial explosion already associated with search in the propositional

C. E. Brown (✉)
Programming Systems Lab, Universität des Saarlandes,
Campus E1 3, 66123 Saarbrücken, Germany
e-mail: cebrown@ps.uni-saarland.de

case. Second, there is the problem of finding the correct instantiations for quantifiers. The instantiation problem appears in the first-order case, but is magnified in the higher-order case by the need to instantiate higher-order variables (e.g., set variables). A third issue that appears in the higher-order case is how one builds in certain basic mathematical properties (e.g., extensionality and choice).

In this paper we give a complete theorem proving procedure for higher-order logic with extensionality and choice. The procedure separates the first issue from the second and third. We start from a complete ground calculus which already builds in extensionality and choice as well as certain restrictions on instantiations. Given a set of formulas to refute, the ground calculus can be used to suggest a sequence of relevant formulas which may be involved in a refutation. The procedure generates propositional clauses corresponding to the meaning of these relevant formulas. When the set of propositional clauses is unsatisfiable (in the propositional sense), then the original set of higher-order formulas is unsatisfiable (in the higher-order Henkin model sense). Conversely, when the original set of higher-order formulas is unsatisfiable, then an unsatisfiable set of propositional clauses will eventually be generated.

Such a procedure has been implemented in the higher-order theorem prover Satallax.[1] The first implementation of Satallax was in Steel Bank Common Lisp. The latest version of Satallax, Satallax 2.4, is implemented in Objective Caml. The SAT-solver MiniSat [12] (coded in C++) is used to determine propositional unsatisfiability.

This paper is a revised and expanded version of [9].

2 Preliminaries

We begin with a brief presentation of Church's simple type theory with a choice operator. For more details see a similar presentation in [3]. Simple types (σ, τ) are given inductively: $o|\iota|\sigma\sigma$. Types $\sigma\tau$ correspond to functions from σ to τ. Terms s, t are generated inductively $x|c|st|\lambda x.s$ where x ranges over variables and c ranges over the logical constants $\bot$, $\rightarrow$, $\forall_\sigma$, $=_\sigma$, $*$ and ε_σ. A name is either a variable or a logical constant. A decomposable name is either a variable or ε_σ for some σ. We use δ to range over decomposable names.

Each variable has a corresponding type σ, and for each type there is a countably infinite set of variables of this type. Likewise each logical constant has a corresponding type: $\bot$ has type o, $\rightarrow$ has type ooo, $*$ has type ι and for each type σ, $\forall_\sigma$ has type $(\sigma o)o$, $=_\sigma$ has type $\sigma\sigma o$ and ε_σ has type $(\sigma o)\sigma$. Note that there are infinitely many logical constants since there are infinitely many types σ. The constant ε_σ is a choice operator at type σ. The constant $*$ plays the role of a "default" element of the nonempty type ι. Types can be assigned to (some) terms in the usual way. From now on we restrict ourselves to typed terms and let Λ_σ be the set of terms of type σ. A *formula* is a term $s \in \Lambda_o$.

We adopt common notational conventions: stu means $(st)u$, $s =_\sigma t$ (or $s = t$) means $=_\sigma st$, $s \rightarrow t$ means $\rightarrow st$, $\neg s$ means $s \rightarrow \bot$, $\top$ means $\neg\bot$, $s \vee t$ means $\neg s \rightarrow t$,

[1]Satallax is available at satallax.com.

$s \wedge t$ means $\neg(s \to \neg t)$, $s \equiv t$ means $(s \to t) \wedge (t \to s)$, $s \neq_\sigma t$ (or $s \neq t$) means $\neg(s =_\sigma t)$, $\forall x.s$ means $\forall_\sigma \lambda x.s$, $\varepsilon x.s$ means $\varepsilon_\sigma \lambda x.s$ and $\exists x.s$ means $\neg\forall x.\neg s$. Binders have as large a scope as is consistent with given parenthesis. For example, in $\forall x.px \to qx$ the occurrence of x in qx is bound by the $\forall$. We will also write abbreviate repeated binders by writing $\lambda x^1 \cdots x^n.s$ for $\lambda x^1. \cdots \lambda x^n.s$ and writing $\forall x^1 \cdots x^n.s$ for $\forall x^1. \cdots \forall x^n.s$. The set $\mathscr{V}t$ of *free variables of t* is defined as usual.

An *accessibility context* ($\mathscr{C}$) is a term with a hole $[]_\sigma$ of the form $[]s_1 \cdots s_n$, $\neg([]s_1 \cdots s_n)$, $([]s_1 \cdots s_n) \neq_\iota s$ or $s \neq_\iota ([]s_1 \cdots s_n)$. We write $\mathscr{C}[s]$ for the term one obtains by putting s into the hole. A term s is *accessible* in a set A of formulas iff there is an accessibility context $\mathscr{C}$ such that $\mathscr{C}[s] \in A$.

Let $[s]$ denote a $\beta\eta$-normal form of s that makes a canonical choice of bound variables. That is, for any $s, t \in \Lambda_\sigma$, $[s] = [t]$ iff s and t are $\alpha\beta\eta$-equivalent. A term s is *normal* if $[s] = s$.

A *substitution* is a type preserving partial function from variables to terms. If θ is a substitution, x is a variable, and s is a term that has the same type as x, we write θ^x_s for the substitution that agrees everywhere with θ except $\theta^x_s x = s$. For each substitution θ let $\hat{\theta}$ be the usual extension of θ to all terms in a capture-avoiding manner.

A *frame* $\mathscr{D}$ is a typed collection of nonempty sets such that $\mathscr{D}_o = \{0, 1\}$ and $\mathscr{D}_{\sigma\tau}$ is a set of total functions from $\mathscr{D}_\sigma$ to $\mathscr{D}_\tau$. An *assignment* $\mathscr{I}$ into $\mathscr{D}$ is a mapping from variables and logical constants of type σ into $\mathscr{D}_\sigma$. An assignment $\mathscr{I}$ is *logical* if it interprets each logical constant to be an element satisfying the corresponding logical property. To be precise, $\mathscr{I}$ is logical if the following properties hold:

- $\mathscr{I}\bot$ is 0.
- For each $a, b \in \mathscr{D}_o$, $\mathscr{I} \to ab$ is 1 iff either a is 0 or b is 1.
- For each type σ and $f \in \mathscr{D}_{\sigma o}$, $\mathscr{I}\forall_\sigma f$ is 1 iff fa is 1 for every $a \in \mathscr{D}_\sigma$.
- For each type σ and $a, b \in \mathscr{D}_\sigma$, $\mathscr{I} =_\sigma ab$ is 1 iff a is b.
- For each type σ and $f \in \mathscr{D}_{\sigma o}$, if there is some $a \in \mathscr{D}_\sigma$ such that fa is 1, then $f(\mathscr{I}\varepsilon_\sigma f)$ is 1.

An assignment $\mathscr{I}$ is an *interpretation* if it can be extended in the usual way to be a total function $\hat{\mathscr{I}}$ mapping each Λ_σ into $\mathscr{D}_\sigma$. A *Henkin model* $(\mathscr{D}, \mathscr{I})$ is a frame $\mathscr{D}$ and a logical interpretation $\mathscr{I}$ into $\mathscr{D}$. We say a formula s is *satisfied* by a Henkin model $(\mathscr{D}, \mathscr{I})$ if $\hat{\mathscr{I}}s = 1$. A set A of formulas is satisfied by a Henkin model if each formula in A is satisfied by the model.

Let A be a set of formulas. A term s is discriminating in A iff there is a term t such that $s \neq_\iota t \in A$ or $t \neq_\iota s \in A$. For each set A of formulas and each type σ we define a nonempty universe $\mathscr{U}^A_\sigma \subseteq \Lambda_\sigma$ as follows.

- Let $\mathscr{U}^A_o = \{\bot, \neg\bot\}$.
- Let $\mathscr{U}^A_\iota$ be the set of discriminating terms in A if there is some discriminating term in A.
- Let $\mathscr{U}^A_\iota = \{*\}$ if there are no discriminating terms in A.
- Let $\mathscr{U}^A_{\sigma\tau} = \{[s] | s \in \Lambda_{\sigma\tau}, \mathscr{V}s \subseteq \mathscr{V}A\}$.

When the set A is clear in context, we write $\mathscr{U}_\sigma$.

A cut-free tableau calculus for higher-order logic with extensionality is given in [10]. The calculus is complete with respect to Henkin models without choice. The details of the completeness proof indicated that one can restrict instantiations for quantifiers on base types to terms occurring on one side of a disequation. This

Fig. 1 Abstract consistency conditions (must hold for every $A \in \Gamma$)

$\mathscr{C}_\bot$	$\bot$ is not in A.
$\mathscr{C}_\neg$	If $\neg s$ is in A, then s is not in A.
$\mathscr{C}_{\neq}$	$s \neq_\iota s$ is not in A.
$\mathscr{C}_\to$	If $s \to t$ is in A, then $A \cup \{\neg s\}$ or $A \cup \{t\}$ is in Γ.
$\mathscr{C}_{\neg\to}$	If $\neg(s \to t)$ is in A, then $A \cup \{s, \neg t\}$ is in Γ.
$\mathscr{C}_\forall$	If $\forall_\sigma s$ is in A, then $A \cup \{[st]\}$ is in Γ for every $t \in \mathscr{U}^A_\sigma$.
$\mathscr{C}_{\neg\forall}$	If $\neg\forall_\sigma s$ is in A, then $A \cup \{\neg[sx]\}$ is in Γ for some variable x.
$\mathscr{C}_{\mathrm{MAT}}$	If $\delta s_1 \ldots s_n$ is in A and $\neg\delta t_1 \ldots t_n$ is in A, then $n \geq 1$ and $A \cup \{s_i \neq t_i\}$ is in Γ for some $i \in \{1,\ldots,n\}$.
$\mathscr{C}_{\mathrm{DEC}}$	If $\delta s_1 \ldots s_n \neq_\iota \delta t_1 \ldots t_n$ is in A, then $n \geq 1$ and $A \cup \{s_i \neq t_i\}$ is in Γ for some $i \in \{1,\ldots,n\}$.
$\mathscr{C}_{\mathrm{CON}}$	If $s =_\iota t$ and $u \neq_\iota v$ are in A, then either $A \cup \{s \neq u, t \neq u\}$ or $A \cup \{s \neq v, t \neq v\}$ is in Γ.
$\mathscr{C}_{\mathrm{BQ}}$	If $s =_o t$ is in A, then either $A \cup \{s, t\}$ or $A \cup \{\neg s, \neg t\}$ is in Γ.
$\mathscr{C}_{\mathrm{BE}}$	If $s \neq_o t$ is in A, then either $A \cup \{s, \neg t\}$ or $A \cup \{\neg s, t\}$ is in Γ.
$\mathscr{C}_{\mathrm{FQ}}$	If $s =_{\sigma\tau} t$ is in A, then $A \cup \{[\forall x. sx =_\tau tx]\}$ is in Γ for some $x \in \mathscr{V}_\sigma \setminus (\mathscr{V}s \cup \mathscr{V}t)$.
$\mathscr{C}_{\mathrm{FE}}$	If $s \neq_{\sigma\tau} t$ is in A, then $A \cup \{\neg[\forall x. sx =_\tau tx]\}$ is in Γ for some $x \in \mathscr{V}_\sigma \setminus (\mathscr{V}s \cup \mathscr{V}t)$.
$\mathscr{C}_\varepsilon$	If $\varepsilon_\sigma s$ is accessible in A, then either $A \cup \{[s(\varepsilon s)]\}$ is in Γ or there is some $x \in \mathscr{V}_\sigma \setminus \mathscr{V}s$ such that $A \cup \{[\forall x. \neg(sx)]\}$ is in Γ.

restriction is shown complete for the first-order case in [10]. The calculus is extended to include choice in [3] and the restriction on instantiations is proven complete in the higher-order case. The proof of completeness makes use of abstract consistency.

We call a finite set of normal formulas a *branch*. A set Γ of branches is an *abstract consistency class* if it satisfies all the conditions in Fig. 1. This definition differs slightly from the one in [3] because we are using $\to$ instead of $\neg$ and $\vee$. With obvious modifications to account for this difference, Theorem 2 in [3] implies that every $A \in \Gamma$ (where Γ is an abstract consistency class) is satisfiable by a Henkin model. We state this here as the *Model Existence Theorem*.

Theorem 1 (Model Existence Theorem) *Let Γ be an abstract consistency class. Each $A \in \Gamma$ is satisfiable by a Henkin model.*

3 Mapping into SAT

We next describe a simple mapping from higher-order formulas into propositional literals and clauses. The essential idea is to abstract away the semantics of logical connectives. The general technique of using a propositional abstraction is standard and is used by SMT solvers (e.g., see [11]).

Let Atom be a countably infinite set of propositional *atoms*. For each atom a, let $\overline{a}$ denote a distinct negated atom. A *literal* is an atom or a negated atom. Let Lit be the set of all literals. Let $\overline{\overline{a}}$ denote a. A *clause* is a finite set of literals, which we write as $l_1 \sqcup \cdots \sqcup l_n$. A *propositional assignment* is a mapping Φ from Atom to $\{0, 1\}$. We extend any such Φ to literals by taking $\Phi(\overline{a}) = 1 - \Phi(a)$. We say an assignment Φ *satisfies a clause* $\mathscr{C}$ if there is some literal $l \in \mathscr{C}$ such that $\Phi l = 1$. An assignment Φ *satisfies a set* $\mathscr{S}$ of clauses if Φ satisfies $\mathscr{C}$ for all $\mathscr{C} \in \mathscr{S}$.

Let $\lfloor . \rfloor$ be a function mapping Λ_o into Lit such that $\lfloor \neg s \rfloor = \overline{\lfloor s \rfloor}$, $\lfloor s \rfloor = \lfloor [s] \rfloor$, and if $\lfloor s \rfloor = \lfloor t \rfloor$, then $\mathscr{I} s = \mathscr{I} t$ in every Henkin model $(\mathscr{D}, \mathscr{I})$.

Remark 1 In the implementation, $\lfloor s \rfloor = \lfloor t \rfloor$ whenever s and t are the same up to $\beta\eta$ and the removal of double negations. Under some flag settings, symmetric equations $u = v$ and $v = u$ are assigned the same literal.

We say Φ is a *pseudo-model* of A if $\Phi \lfloor s \rfloor = 1$ for all $s \in A$. We say an assignment Φ is *Henkin consistent* if there is a Henkin model $(\mathscr{D}, \mathscr{I})$ such that $\Phi \lfloor s \rfloor = \hat{\mathscr{I}} s$ for all $s \in \Lambda_o$.

4 States and Successors

Definition 1 A *quasi-state* Σ is a 5-tuple $(F_p^\Sigma, F_a^\Sigma, U_p^\Sigma, U_a^\Sigma, C^\Sigma)$ where F_p^Σ and F_a^Σ are finite sets of normal formulas, U_p^Σ and U_a^Σ are finite sets of normal terms, and C^Σ is a finite set of clauses. We call formulas in F_p^Σ *passive formulas*, formulas in F_a^Σ *active formulas*, terms in U_p^Σ *passive instantiations* and terms in U_a^Σ *active instantiations*.

Given a quasi-state Σ, we define the following notation:

$$F^\Sigma := F_p^\Sigma \cup F_a^\Sigma \qquad U^\Sigma := U_p^\Sigma \cup U_a^\Sigma \qquad U_{p,\sigma}^\Sigma := U_p^\Sigma \cap \Lambda_\sigma \qquad U_{a,\sigma}^\Sigma := U_a^\Sigma \cap \Lambda_\sigma$$

During the procedure, we will only consider quasi-states that satisfy certain invariants. Such a quasi-state will be called a *state*. Before giving the technical definition of a state, we consider a simple example. In this example we will refer to the quasi-states as *states*, as they will always satisfy the relevant properties.

Each step of the search process will pass from one state to a successor state. The passive formulas and passive instantiations of a successor state will always include all the passive formulas and passive instantiations of the previous state. Likewise, all the clauses of the previous state will be clauses of the successor state. Often we obtain a successor state by moving an active formula (instantiation) to the set of passive formulas (instantiations). We will refer to this as *processing* the formula (instantiation).

This use of the terminology "active" and "passive" was introduced in [9]. If a formula or instantiation is active, it is waiting to be processed which will cause some action to be taken. An alternative would be to refer to active formulas and instantiations as "pending." If a formula or instantiation is passive, then it has already been processed and will no longer directly cause any actions. Passive formulas and instantiations can be used as side information when processing active formulas and instantiations. An alternative would be to refer to passive formulas and instantiations as "usable." To avoid confusion, we point out that in some theorem proving literature (e.g., the description of Inst-Gen in [14]) the adjectives "active" and "passive" are used in just the opposite way: a formula is "passive" if it is waiting to be processed and "active" after having been processed.

Example 1 Let p, q be variables of type o. Suppose we wish to refute the branch with two formulas: p and $\forall q.p \to q$. We begin with a state Σ_0 with $F_p^{\Sigma_0} = \emptyset$, $F_a^{\Sigma_0} =$

$\{p, \forall q.p \to q\}$, $U_p^{\Sigma_0} = \{\bot, \top\}$, $U_a^{\Sigma_0} = \emptyset$ and C^{Σ_0} contains exactly the two unit clauses $\lfloor p \rfloor$ and $\lfloor \forall q.p \to q \rfloor$. We will refute this branch in one step. In particular, we process the formula $\forall q.p \to q$ by moving it from being active to passive and by applying all the instantiations of type o in $U_p^{\Sigma_0}$. This results in a state Σ_1 in which $F_p^{\Sigma_1} = \{\forall q.p \to q\}$, $F_a^{\Sigma_1} = \{p, p \to \bot, p \to \top\}$, $U_p^{\Sigma_1} = U_p^{\Sigma_0}$, $U_a^{\Sigma_1} = U_a^{\Sigma_0}$ and C^{Σ_1} contains the two unit clauses from C^{Σ_0} as well as the two clauses $\overline{\lfloor \forall q.p \to q \rfloor} \sqcup \lfloor p \to \bot \rfloor$ and $\overline{\lfloor \forall q.p \to q \rfloor} \sqcup \lfloor p \to \top \rfloor$. Note that $\lfloor p \to \bot \rfloor$ is the same as $\overline{\lfloor p \rfloor}$. Clearly there is no propositional assignment satisfying the clauses in C^{Σ_1}. This completes the refutation. The two states can be displayed as in Fig. 2. Since the sets F_p, U_p and C grow as we transition from one state to the next, the members of $F_p^{\Sigma_1}$, $U_p^{\Sigma_1}$ and C^{Σ_1} are the ones listed in the appropriate column either in the Σ_0 row or the Σ_1 row. That is, in the Σ_1 row, we only list the new members. For example, C^{Σ_1} consists of four clauses: the two unit clauses in C^{Σ_0} and the two new clauses $\overline{\lfloor \forall q.p \to q \rfloor} \sqcup \overline{\lfloor p \rfloor}$ and $\overline{\lfloor \forall q.p \to q \rfloor} \sqcup \lfloor p \to \top \rfloor$ listed in the entry for row Σ_1 and column C. On the other hand, we will both add and delete members from F_a and U_a as we transition from one state to the next. We will continue to only list the new members in the appropriate row and column, and indicate which members are deleted by crossing the member out of the appropriate row and column. For example, $F_a^{\Sigma_0}$ contains two formulas p and $\forall q.p \to q$ and $F_a^{\Sigma_1}$ still contains p, no longer contains $\forall q.p \to q$, and contains the new formulas $p \to \bot$ and $p \to \top$. Accordingly, we indicate the members of $F_a^{\Sigma_1}$ by the entries $\cancel{\forall q.p \to q}$, $p \to \bot$ and $p \to \top$ in the entry for row Σ_1 and column F_a.

Definition 2 A quasi-state $\Sigma = (F_p^{\Sigma}, F_a^{\Sigma}, U_p^{\Sigma}, U_a^{\Sigma}, C^{\Sigma})$ is a *state* if the conditions in Fig. 3 hold and for every clause $\mathscr{C}$ in C^{Σ} and every literal $l \in \mathscr{C}$, either $l = \lfloor s \rfloor$ for some $s \in F^{\Sigma}$ or $l = \overline{\lfloor s \rfloor}$ for some $s \in F_p^{\Sigma}$. A variable x is *fresh* for a state Σ if x is not free in any $s \in F^{\Sigma} \cup U^{\Sigma}$.

We say a propositional assignment Φ *s*atisfies a state Σ if Φ satisfies C^{Σ}. We say Σ is *propositionally satisfiable* if there is a Φ such that Φ satisfies Σ. Otherwise, we say Σ is *propositionally unsatisfiable*. Furthermore, we say Σ is *Henkin satisfiable* if there is a Henkin consistent propositional assignment satisfying C^{Σ}. Note that checking whether Σ is propositionally satisfiable is simply a SAT-problem.

We now consider a second simple example.

Example 2 Let p be a variable of type ιo and x be a variable of type ι. Suppose we wish to prove the following basic property of the choice operator ε_ι: $\forall x.px \to p(\varepsilon_\iota p)$. The refutation will proceed in seven steps taking us from an initial state Σ_0

	F_p	F_a	U_p	U_a	C
Σ_0		$p, \forall q.p \to q$	$\bot, \top$		$\lfloor p \rfloor$ $\lfloor \forall q.p \to q \rfloor$
Σ_1	$\forall q.p \to q$	$\cancel{\forall q.p \to q}$ $p \to \bot, p \to \top$			$\overline{\lfloor \forall q.p \to q \rfloor} \sqcup \overline{\lfloor p \rfloor}$ $\overline{\lfloor \forall q.p \to q \rfloor} \sqcup \lfloor p \to \top \rfloor$

Fig. 2 States from Example 1

$\mathscr{S}_\bot$	If $\bot$ is in F_p, then $\overline{\lfloor\bot\rfloor}$ is in C.
$\mathscr{S}_{\neq}$	If $s \neq_\iota s$ is in F_p, then $\lfloor s = s\rfloor$ is in C.
$\mathscr{S}_{\to}$	If $s \to t$ is in F_p and t is not $\bot$, then $\{\neg s, t\} \subseteq F$ and $\overline{\lfloor s \to t\rfloor} \sqcup \lfloor\neg s\rfloor \sqcup \lfloor t\rfloor$ is in C.
$\mathscr{S}_{\neg\to}$	If $\neg(s \to t)$ is in F_p, then $\{s, \neg t\} \subseteq F$, $\lfloor s \to t\rfloor \sqcup \overline{\lfloor s\rfloor}$ and $\lfloor s \to t\rfloor \sqcup \lfloor\neg t\rfloor$ are in C.
$\mathscr{S}_\forall$	If $\forall_\sigma s$ is in F_p and $t \in U_{p,\sigma}$, then $[st] \in F$ and $\overline{\lfloor\forall_\sigma s\rfloor} \sqcup \lfloor st\rfloor$ is in C.
$\mathscr{S}_{\neg\forall}$	If $\neg\forall_\sigma s$ is in F_p, then there is some variable x of type σ such that $\neg[sx] \in F$ and $\lfloor\forall_\sigma s\rfloor \sqcup \overline{\lfloor sx\rfloor}$ is in C.
$\mathscr{S}_{\mathrm{MAT}}$	If $\delta s_1 \dots s_n$ and $\neg\delta t_1 \dots t_n$ are in F_p where $n \geq 1$, then $s_i \neq t_i$ is in F for each $i \in \{1, \dots, n\}$ and $\overline{\lfloor\delta s_1 \dots s_n\rfloor} \sqcup \lfloor\delta t_1 \dots t_n\rfloor \sqcup \lfloor s_1 \neq t_1\rfloor \sqcup \dots \sqcup \lfloor s_n \neq t_n\rfloor$ is in C.
$\mathscr{S}_{\mathrm{DEC}}$	If $\delta s_1 \dots s_n \neq_\iota \delta t_1 \dots t_n$ is in F_p where $n \geq 1$, then $s_i \neq t_i$ is in F for each $i \in \{1, \dots, n\}$ and $\lfloor\delta s_1 \dots s_n = \delta t_1 \dots t_n\rfloor \sqcup \lfloor s_1 \neq t_1\rfloor \sqcup \dots \sqcup \lfloor s_n \neq t_n\rfloor$ is in C.
$\mathscr{S}_{\mathrm{CON}}$	If $s =_\iota t$ and $u \neq_\iota v$ are in F_p, then $\{s \neq u, t \neq u, s \neq v, t \neq v\} \subseteq F$ and the following four clauses are in C: $\overline{\lfloor s = t\rfloor} \sqcup \lfloor u = v\rfloor \sqcup \lfloor s \neq u\rfloor \sqcup \lfloor s \neq v\rfloor$, $\overline{\lfloor s = t\rfloor} \sqcup \lfloor u = v\rfloor \sqcup \lfloor s \neq u\rfloor \sqcup \lfloor t \neq v\rfloor$, $\overline{\lfloor s = t\rfloor} \sqcup \lfloor u = v\rfloor \sqcup \lfloor t \neq u\rfloor \sqcup \lfloor s \neq v\rfloor$, $\overline{\lfloor s = t\rfloor} \sqcup \lfloor u = v\rfloor \sqcup \lfloor t \neq u\rfloor \sqcup \lfloor t \neq v\rfloor$
$\mathscr{S}_{\mathrm{BQ}}$	If $s =_o t$ is in F_p, then $\{s, t, \neg s, \neg t\} \subseteq F$ and $\overline{\lfloor s = t\rfloor} \sqcup \lfloor s\rfloor \sqcup \lfloor\neg t\rfloor$ and $\overline{\lfloor s = t\rfloor} \sqcup \lfloor\neg s\rfloor \sqcup \lfloor t\rfloor$ are in C.
$\mathscr{S}_{\mathrm{BE}}$	If $s \neq_o t$ is in F_p, then $\{s, t, \neg s, \neg t\} \subseteq F$ and $\lfloor s = t\rfloor \sqcup \lfloor s\rfloor \sqcup \lfloor t\rfloor$ and $\lfloor s = t\rfloor \sqcup \lfloor\neg s\rfloor \sqcup \lfloor\neg t\rfloor$ are in C.
$\mathscr{S}_{\mathrm{FQ}}$	If $s =_{\sigma\tau} t$ is in F_p, then there is some $x \in \mathscr{V}_\sigma \setminus (\mathscr{V}s \cup \mathscr{V}t)$ such that $[\forall x.sx =_\tau tx]$ is in F and $\overline{\lfloor s = t\rfloor} \sqcup \lfloor\forall x.sx = tx\rfloor$ is in C.
$\mathscr{S}_{\mathrm{FE}}$	If $s \neq_{\sigma\tau} t$ is in F_p, then there is some $x \in \mathscr{V}_\sigma \setminus (\mathscr{V}s \cup \mathscr{V}t)$ such that $[\neg\forall x.sx =_\tau tx]$ is in F and $\lfloor s = t\rfloor \sqcup \lfloor\neg\forall x.sx = tx\rfloor$ is in C.
$\mathscr{S}_\varepsilon$	If $\varepsilon_\sigma s$ is accessible in F_p, then there is some $x \in \mathscr{V}_\sigma \setminus \mathscr{V}s$ such that $[s(\varepsilon s)]$ and $[\forall x.\neg(sx)]$ are in F and $\lfloor s(\varepsilon s)\rfloor \sqcup \lfloor\forall x.\neg(sx)\rfloor$ is in C.

Fig. 3 Conditions on a quasi-state $\Sigma = (F_p, F_a, U_p, U_a, C)$

(corresponding to assuming the negation) to a state Σ_7 such that C^{Σ_7} is propositionally unsatisfiable. The states Σ_i for $i \in \{0, \dots, 7\}$ are indicated in Fig. 4. In the first step we process $\neg\forall x.px \to p(\varepsilon p)$ by choosing a fresh variable y of type ι and including the new formula $\neg(py \to p(\varepsilon p))$ and a clause relating the literals

	F_p	F_a	U_p	U_a	C
Σ_0		$\neg\forall x.px \to p(\varepsilon p)$			$\overline{\lfloor\forall x.px \to p(\varepsilon p)\rfloor}$
Σ_1	$\neg\forall x.px \to p(\varepsilon p)$	$\cancel{\neg\forall x.px \to p(\varepsilon p)}$ $\neg(py \to p(\varepsilon p))$			$\lfloor\forall x.px \to p(\varepsilon p)\rfloor \sqcup \overline{\lfloor py \to p(\varepsilon p)\rfloor}$
Σ_2	$\neg(py \to p(\varepsilon p))$	$\cancel{\neg(py \to p(\varepsilon p))}$ py, $\neg p(\varepsilon p)$			$\lfloor py \to p(\varepsilon p)\rfloor \sqcup \lfloor py\rfloor$ $\lfloor py \to p(\varepsilon p)\rfloor \sqcup \overline{\lfloor p(\varepsilon p)\rfloor}$
Σ_3	py	$\cancel{py}$			
Σ_4	$\neg(p(\varepsilon p))$	$\cancel{\neg p(\varepsilon p)}$ $y \neq \varepsilon p$			$\overline{\lfloor py\rfloor} \sqcup \lfloor p(\varepsilon p)\rfloor \sqcup \overline{\lfloor y = \varepsilon p\rfloor}$
Σ_5	$y \neq \varepsilon p$	$\cancel{y \neq \varepsilon p}$ $\forall x.\neg px$, $p(\varepsilon p)$		y, εp	$\lfloor p(\varepsilon p)\rfloor \sqcup \lfloor\forall x.\neg px\rfloor$
Σ_6	$\forall x.\neg px$	$\cancel{\forall x.\neg px}$, $p(\varepsilon p)$			
Σ_7		$\neg py$	y	$\cancel{y}$	$\overline{\lfloor\forall x.\neg px\rfloor} \sqcup \overline{\lfloor py\rfloor}$

Fig. 4 States from Example 2

corresponding to the two formulas. The resulting state is Σ_1. Note that Σ_1 satisfies the condition $\mathscr{S}_{\neg\forall}$. We obtain Σ_2 by processing $\neg(py \to p(\varepsilon p))$ and obtaining two new formulas py and $\neg p(\varepsilon p)$ and two new clauses. Note that Σ_2 satisfies the condition $\mathscr{S}_{\neg\to}$. We obtain Σ_3 by processing py. In general, processing such a formula involves mating it with all passive formulas of the form $\neg pt$ (in order to ensure $\mathscr{S}_{\text{MAT}}$ holds). Since there are no such *passive* formulas (in particular, $\neg p(\varepsilon p)$ is active), Σ_3 only differs from Σ_2 in that py has been made passive. We obtain Σ_4 by processing $\neg p(\varepsilon p)$ and ensuring $\mathscr{S}_{\text{MAT}}$ holds. This involves mating it with the passive formula py to obtain the formula $y \neq \varepsilon p$ and adding a new clause. (The reader should note that the new clause in Σ_4 will not be used to show the final set of clauses is propositionally unsatisfiable.) To obtain Σ_5 we process $y \neq \varepsilon p$. Since y and εp are discriminating terms in the set of passive formulas of Σ_5, we add them to the set of active instantiations (though no condition in Fig. 3 requires this). Also, since εp is accessible in $F_p^{\Sigma_5}$, we include the formulas $\forall x.\neg px$ and $p(\varepsilon p)$ as well as a clause corresponding to the meaning of the choice operator ε in order to ensure $\mathscr{S}_\varepsilon$ holds. We obtain Σ_6 by processing $\forall x.\neg px$. In principle, this means instantiating with all passive instantiations of type ι (to ensure $\mathscr{S}_\forall$ holds), but we have no *passive* instantiations of this type. Finally, we obtain Σ_7 by processing the instantiation y. Since y has type ι, we will use it as an instantiation for the passive formula $\forall x.\neg px$ (to ensure $\mathscr{S}_\forall$ holds). As a consequence, we add the formula $\neg py$ and a corresponding clause. At this point, the clauses are propositionally unsatisfiable and we are done.

Given a branch A, an *initial state* Σ *for* A is a state with $A \subseteq F^\Sigma$, and $C^\Sigma = \{\lfloor s \rfloor | s \in A\}$. (We require $A \subseteq F^\Sigma$ rather than $A \subseteq F_a^\Sigma$ to allow for the possibility that some formulas in A are passive rather than active in an initial state. In practice, this could result from some preprocessing of formulas in A.) To see that for any branch A there is an initial state, consider Σ with $F_p^\Sigma = \emptyset$, $F_a^\Sigma = A$, $U_p^\Sigma = \emptyset$, $U_a^\Sigma = \emptyset$ and $C^\Sigma = \{\lfloor s \rfloor | s \in A\}$.

Definition 3 We say a state Σ' is a *successor* of a state Σ (and write $\Sigma \to \Sigma'$) if $F_p^\Sigma \subseteq F_p^{\Sigma'}$, $F_a^\Sigma \subseteq F^{\Sigma'}$, $U_p^\Sigma \subseteq U_p^{\Sigma'}$, $U_a^\Sigma \subseteq U^{\Sigma'}$, $C^\Sigma \subseteq C^{\Sigma'}$ and if Σ is Henkin satisfiable, then Σ' is Henkin satisfiable.

Note that the successor relation is reflexive and transitive. Also, soundness of the procedure is built into the definition of the successor relation.

Proposition 1 (Soundness) *Let A be a branch. If there is a propositionally unsatisfiable Σ' such that $\Sigma_A \to \Sigma'$, then A is unsatisfiable.*

Proof Assume $(\mathscr{D}, \mathscr{I})$ is a Henkin model of A. Choose Φ such that $\Phi\lfloor s \rfloor = \hat{\mathscr{I}} s$ for each $s \in A$. Clearly, Φ demonstrates that Σ_A is Henkin satisfiable. On the other hand, since Σ' is propositionally unsatisfiable, it is Henkin unsatisfiable. This contradicts the definition of $\Sigma_A \to \Sigma'$. □

A strategy which chooses a successor state for each propositionally satisfiable state will yield a sound procedure. One such strategy is to interleave two kinds of actions: (1) process active formulas and instantiations while making the minimal number of additions of formulas and clauses consistent with the invariants in Fig. 3

and (2) generate new active instantiations. To ensure soundness, when processing a formula $\overline{\neg\forall_\sigma s}$ a procedure should choose a fresh variable x, add $\neg[sx]$ to F_a and add $\lfloor\forall_\sigma s\rfloor \sqcup \overline{\lfloor sx\rfloor}$ to C.

If a strategy does not lead to a propositionally unsatisfiable state, then it will give a finite or infinite path of states. If the strategy is fair, this path will satisfy certain fairness properties. In this case, we can use the path to prove the original branch is satisfiable. That is, we can conclude that every fair strategy is complete.

Definition 4 Let $\alpha \in \omega \cup \{\omega\}$. An *$\alpha$-path* (or, simply *path*) is an α-sequence $\overline{\Sigma} = (\Sigma_i)_{i<\alpha}$ of propositionally satisfiable states such that $\Sigma_i \to \Sigma_{i+1}$ for each i with $i+1<\alpha$. We say a type σ is a *quantified type* on the path if there exist $i<\alpha$ and s such that $\forall_\sigma s \in F^{\Sigma_i}$. An α-path is *fair* if the following conditions hold:

1. For all $i<\alpha$ and $s \in F_a^{\Sigma_i}$ there is some $j \in [i,\alpha)$ such that $s \in F_p^{\Sigma_j}$. (Every active formula eventually gets processed.)
2. If σ is a quantified type, then for all $i<\alpha$, $A \subseteq F^{\Sigma_i}$ and $t \in \mathscr{U}_\sigma^A$ there is some $j \in [i,\alpha)$ such that $t \in U_p^{\Sigma_j}$. (Every potential instantiation eventually gets processed.)

Given a branch A_0, we will start with an initial state Σ_0 for A_0. Our theorem proving procedure will construct a sequence of successor states in such a way that, unless some state is propositionally unsatisfiable, the sequence will be a fair path. In order to prove completeness of this procedure, it is enough to prove that if there is a fair path starting from Σ_0, then A_0 is satisfiable. This result will be given as Theorem 2 at the end of this section.

For the remainder of this section we assume a fixed α and fair α-path $\overline{\Sigma}$.

Definition 5 Let $i<\alpha$ be given. We say a branch A is *i-supported* if $A \subseteq F^{\Sigma_i}$ and there is a pseudo-model Φ of A satisfying Σ_i. We say a branch A is *i-consistent* if A is j-supported for all $j \in [i,\alpha)$.

Lemma 1 *Let $i<\alpha$ and $j \in [i,\alpha)$ be given. If A is j-supported and $A \subseteq F^{\Sigma_i}$, then A is i-supported.*

Proof This follows from $C^{\Sigma_i} \subseteq C^{\Sigma_j}$. □

Let Γ be the set of all branches A such that A is i-consistent for some $i<\alpha$. We will prove Γ is an abstract consistency class.

Lemma 2 *Let A be a j-consistent branch. Let $A_1,\ldots,A_n$ be branches such that $A \subseteq A_l \subseteq F^{\Sigma_j}$ for each $l \in \{1,\ldots,n\}$. Either there is some $l \in \{1,\ldots,n\}$ such that A_l is j-consistent or there is some $k \in \left[j,\alpha\right)$ such that A_l is not k-supported for each $l \in \{1,\ldots,n\}$.*

Proof Assume none of $A_1,\ldots,A_n$ is j-consistent. Let $k_1,\cdots,k_n \in \left[j,\alpha\right)$ be such that A_l is not k_l-supported for each $l \in \{1,\ldots,n\}$. Let k be the maximum of $k_1,\ldots,k_n$. By Lemma 1 each A_l is not k-supported. □

Lemma 3 *Γ is an abstract consistency class.*

Proof We verify a representative collection of cases.

$\mathscr{C}_\bot$ Suppose $\bot \in A$ and A is i-consistent. By fairness there is some $j \in [i, \alpha)$ such that $\bot \in F_p^{\Sigma_j}$. By $\mathscr{S}_\bot$ the unit clause $\overline{\lfloor\bot\rfloor}$ is in C^{Σ_j}. This contradicts A being j-supported.

$\mathscr{C}_\neg$ Suppose $\neg s$ and s are in A. Since no propositional assignment Φ can have $\Phi\lfloor\neg s\rfloor = 1$ and $\Phi\lfloor s\rfloor = 1$, A cannot be i-consistent for any i.

$\mathscr{C}_\rightarrow$ Suppose $s \rightarrow t$ is in an i-consistent branch A. If t is $\bot$, then $A \cup \{\neg s\}$ is the same as A and so $A \cup \{\neg s\}$ is i-consistent. Assume t is not $\bot$. Since A is i-consistent, we know $A \subseteq F^{\Sigma_i}$ and so $s \rightarrow t \in F^{\Sigma_i}$. By fairness there is some $j \in [i, \alpha)$ such that $s \rightarrow t \in F_p^{\Sigma_j}$. By $\mathscr{S}_\rightarrow$ we know $\{\neg s, t\} \subseteq F^{\Sigma_j}$ and $\overline{\lfloor s \rightarrow t\rfloor} \sqcup \overline{\lfloor s\rfloor} \sqcup \lfloor t\rfloor$ is in C^{Σ_j}. Note that $A \cup \{\neg s\} \subseteq F^{\Sigma_k}$ and $A \cup \{t\} \subseteq F^{\Sigma_k}$ for every $k \in \left[j, \alpha\right)$. Assume neither $A \cup \{\neg s\}$ nor $A \cup \{t\}$ is j-consistent. By Lemma 2 there is some $k \in \left[j, \alpha\right)$ such that neither $A \cup \{\neg s\}$ nor $A \cup \{t\}$ is k-supported. Since A is i-consistent, A is k-supported and has some pseudo-model Φ satisfying Σ_k. Since $\overline{\lfloor s \rightarrow t\rfloor} \sqcup \overline{\lfloor s\rfloor} \sqcup \lfloor t\rfloor$ is in C^{Σ_k} and $\Phi\lfloor s \rightarrow t\rfloor = 1$, we must have $\Phi\lfloor s\rfloor = 0$ or $\Phi\lfloor t\rfloor = 1$. Thus Φ witnesses that either $A \cup \{\neg s\}$ or $A \cup \{t\}$ is k-supported, contradicting our choice of k. Hence either $A \cup \{\neg s\}$ or $A \cup \{t\}$ must be j-consistent.

$\mathscr{C}_{\neg\rightarrow}$ Suppose $\neg(s \rightarrow t)$ is in an i-consistent branch A. Since A is i-consistent, we know $\neg(s \rightarrow t) \in F^{\Sigma_i}$. By fairness there is some $j \in [i, \alpha)$ such that $\neg(s \rightarrow t) \in F_p^{\Sigma_j}$. By $\mathscr{S}_{\neg\rightarrow}$ we know $\{s, \neg t\} \subseteq F^{\Sigma_j}$, and both $\lfloor s \rightarrow t\rfloor \sqcup \lfloor s\rfloor$ and $\lfloor s \rightarrow t\rfloor \sqcup \overline{\lfloor t\rfloor}$ are in C^{Σ_j}. We prove $A \cup \{s, \neg t\}$ is j-consistent. Let $k \in \left[j, \alpha\right)$ be given. Since A is i-consistent, it has some pseudo-model Φ satisfying Σ_k. Since $\Phi\lfloor\neg(s \rightarrow t)\rfloor = 1$, we must have $\Phi\lfloor s\rfloor = 1$ and $\Phi\lfloor\neg t\rfloor = 1$. Hence Φ is a pseudo-model of $A \cup \{s, \neg t\}$ and so $A \cup \{s, \neg t\}$ is k-supported. Therefore, $A \cup \{s, \neg t\}$ is j-consistent.

$\mathscr{C}_\forall$ Let A be an i-consistent branch such that $\forall_\sigma s \in A$ and $t \in \mathscr{U}_\sigma^A$. Note that $\forall_\sigma s \in A \subseteq F^{\Sigma_i}$ witnesses that σ is a quantified type on the path. By fairness there is some $j \in [i, \alpha)$ such that $\forall s \in F_p^{\Sigma_j}$ and $t \in U_p^{\Sigma_j}$. By $\mathscr{S}_\forall$ $[st] \in F^{\Sigma_j}$ and $\overline{\lfloor\forall_\sigma s\rfloor} \sqcup \lfloor st\rfloor$ is in C^{Σ_j}. We prove A is j-consistent. Let $k \in \left[j, \alpha\right)$ be given. Since A is i-consistent, it has some pseudo-model Φ satisfying Σ_k. Since $\Phi\lfloor\forall s\rfloor = 1$ and $\overline{\lfloor\forall_\sigma s\rfloor} \sqcup \lfloor st\rfloor$ is in C^{Σ_j}, we must have $\Phi\lfloor st\rfloor = 1$ and so $A \cup \{[st]\}$ is k-supported. (We know $\lfloor[st]\rfloor = \lfloor st\rfloor$ as a property of $\lfloor\cdot\rfloor$.)

$\mathscr{C}_{\neg\forall}$ Let A be an i-consistent branch such that $\neg\forall_\sigma s \in A$. By fairness there is some $j \in [i, \alpha)$ such that $\neg\forall s \in F_p^{\Sigma_j}$. By $\mathscr{S}_{\neg\forall}$ there is some variable x such that $\neg[sx] \in F^{\Sigma_j}$ and $\lfloor\forall_\sigma s\rfloor \sqcup \overline{\lfloor sx\rfloor}$ is in C^{Σ_j}. Let $k \in \left[j, \alpha\right)$ be given. Let Φ be a pseudo-model of A satisfying Σ_k. Since $\Phi\lfloor\neg\forall s\rfloor = 1$ we must have $\Phi\lfloor\neg(sx)\rfloor = 1$ and so $A \cup \{\neg[sx]\}$ is k-supported.

$\mathscr{C}_{\mathrm{CON}}$ Suppose $s =_\iota t$ and $u \neq_\iota v$ are in an i-consistent branch A. By fairness there is some $j \in [i, \alpha)$ such that $s =_\iota t$ and $u \neq_\iota v$ are $F_p^{\Sigma_j}$. By $\mathscr{S}_{\mathrm{CON}}$ we know the formulas $s \neq u, t \neq u, s \neq v$ and $t \neq v$ are in F^{Σ_j} and the following four clauses are in C^{Σ_j}:

$$\overline{\lfloor s = t\rfloor} \sqcup \lfloor u = v\rfloor \sqcup \lfloor s \neq u\rfloor \sqcup \lfloor s \neq v\rfloor,\ \overline{\lfloor s = t\rfloor} \sqcup \lfloor u = v\rfloor \sqcup \lfloor s \neq u\rfloor \sqcup \lfloor t \neq v\rfloor$$

$$\overline{\lfloor s = t\rfloor} \sqcup \lfloor u = v\rfloor \sqcup \lfloor t \neq u\rfloor \sqcup \lfloor s \neq v\rfloor,\ \overline{\lfloor s = t\rfloor} \sqcup \lfloor u = v\rfloor \sqcup \lfloor t \neq u\rfloor \sqcup \lfloor t \neq v\rfloor$$

Assume neither $A \cup \{s \neq u, t \neq u\}$ nor $A \cup \{s \neq v, t \neq v\}$ is j-consistent. We know by Lemma 2 that there is some $k \in [j, \alpha)$ such that neither $A \cup \{s \neq u, t \neq u\}$ nor $A \cup \{s \neq v, t \neq v\}$ is k-supported. Let Φ be a pseudo-model of A satisfying Σ_k. Note that $\Phi\lfloor s = t\rfloor = 1$ and $\Phi\lfloor u = v\rfloor = 0$. By examining the four clauses above, it is clear that we must either have $\Phi\lfloor s \neq u\rfloor = 1$ and $\Phi\lfloor t \neq u\rfloor = 1$ or have $\Phi\lfloor s \neq v\rfloor = 1$ and $\Phi\lfloor t \neq v\rfloor = 1$, a contradiction. □

Theorem 2 (Model Existence) *Let A_0 be a branch and $\overline{\Sigma}$ be a fair α-path such that Σ_0 is an initial state for Σ_{A_0}. Then A_0 is satisfiable.*

Proof By Theorem 1 it is enough to prove A_0 is 0-consistent. Let $j \in [0, \alpha)$ be given. Clearly $A_0 \subseteq F^{\Sigma_0} \subseteq F^{\Sigma_j}$. Let Φ satisfy Σ_j. For each $s \in A_0$, the unit clause $\lfloor s\rfloor$ is in C^{Σ_j} and so $\Phi\lfloor s\rfloor = 1$. □

5 Implementation

A procedure along the lines described above has been implemented in a theorem prover named Satallax. There are some minor differences from the abstract description. One difference is that double negations are eliminated during normalization in the implementation (e.g., the normal form of $p(\lambda x.\neg\neg x)$ is $p(\lambda x.x)$). Another difference is that there is no default constant $*$ of type ι. If there are no discriminating terms of type ι, then either a variable or the term $\varepsilon_\iota x.\bot$ is used as an instantiation of type ι. Also, there may be base types other than ι.

The first version of Satallax was written in Steel Bank Common Lisp. In this earlier version, MiniSat was restarted and sent all the clauses generated so far whenever propositional satisfiability was to be tested. The latest version of Satallax is implemented in Objective Caml. A foreign function interface allows Satallax to call MiniSat functions (coded in C++) in order to add new clauses to the current set of clauses and to test for satisfiability of the current set of clauses. This is a much more efficient way of using MiniSat.

Problems are given to Satallax as a TPTP file in THF format [22]. We will refer to several problems from the TPTP problem library [19] (specifically, TPTP v5.3.0). The Objective Caml code to support parsing of TPTP problems originally written for LEO-II [5] was incorporated into Satallax with minor modifications. A TPTP problem file may include axioms and optionally a conjecture. The conjecture, if given, is negated and treated as an axiom. Logical constants that occur in axioms are rewritten in favor of the basic logical constants $\bot$, $\rightarrow$, $=_\sigma$, $\forall_\sigma$ and ε_σ. Also, all definitions are expanded and the terms are $\beta\eta$-normalized. De Bruijn indices are used to deal with α-convertibility. If the normalized axiom s is of the particular form $\forall px.px \rightarrow p(ep)$ or $\forall p.(\neg\forall x.\neg px) \rightarrow p(ep)$ where e is a constant of type $(\sigma o)\sigma$ for some σ, then e is registered as a choice operator of type σ and the axiom s is omitted from the initial branch. Every other normalized axiom is an initial assumption. The choice rule can be applied with every name registered as a choice operator.

There are about a hundred flags that can be set in order to control the order in which the search space is explored. A collection of flag settings is called a *mode*. Currently, there are about 300 modes predefined in Satallax. A particular mode can be chosen via a command line option. Otherwise, a default schedule of modes is used and each of the modes on the schedule is given a certain amount of time to try to refute the problem.

If the flag SPLIT_GLOBAL_DISJUNCTIONS is set to TRUE, then Satallax will decompose the topmost logical connectives including the topmost disjunctions. This is likely to result in a set of subgoals which can be solved independently. For example, if the problem consists of one axiom of the form $(p \vee q) \wedge r$ and a conjecture is of the form $r \wedge (q \vee p)$ (where p, q, and r are variables of type o), then Satallax will decompose this into four subgoals (i.e., four branches) to refute independently. These subgoals will be

1. $p, r, \neg r$,
2. $q, r, \neg r$,
3. $p, r, \neg q, \neg p$ and
4. $q, r, \neg q, \neg p$.

Doing such a decomposition is an especially good idea if, for example, the conjecture is a conjunction. On the other hand, doing such a decomposition is, of course, a bad idea if there are many disjunctive axioms.

Once the initial branch is determined, the state is initialized to include a unit clause for each member and the set of active formulas is initialized to be the initial branch. The terms $\perp$ and $\neg\perp$ are added as passive instantiations. Additionally, if the flag INITIAL_SUBTERMS_AS_INSTANTIATIONS is set to TRUE, then all subterms of the initial branch are added as passive instantiations. During the search, discriminating terms of type ι are added as active instantiations. If there is a quantifier at a function type $\sigma\tau$, a process of enumerating normal terms of type $\sigma\tau$ is started. Of course, this enumeration process is the least directed part of the search procedure.

The successor relation on states was defined very generally. In particular, it does not rule out adding more formulas, instantiations and clauses than the ones suggested by the invariants on states. These additions may be very useful, but they are not necessary for completeness. A simple example is that, if the flag INSTANTIATE_WITH_FUNC_DISEQN_SIDES is set to TRUE, the terms s and t are added as active instantiations whenever an active formula $s \neq_{\sigma\tau} t$ is processed. In addition, if HOUNIF1 is set to TRUE, then (bounded) higher-order unification is used to suggest terms to add as active instantiations.

Example 3 The conjecture of the problem SEU868^5 from the TPTP states that one characterization of finiteness of a set implies another characterization. The origin of the problem is THM551 in the library of TPS [2] where it is formulated using a number of (polymorphic) definitions. Let σ be a type. One can think of terms of type σo as representing sets over type σ via characteristic functions. In this way, we think of $\lambda x.\perp$ (where x has type σ) as corresponding to the empty set. Also, if we are given a "set" p of type σo and an "element" x of type σ, then we can represent the set consisting of the elements of p as well as the (possibly) new element x by the term $\lambda t.p\,t \vee t = x$ (where t is a variable of type σ). In the TPS library this operation of adding one element to a set is given as a polymorphic definition ADD1 of type $(\sigma o)\sigma\sigma o$ which is defined to be the term $\lambda pxt.p\,t \vee t = x$ where p has type σo and x and t have type σ. A natural way to define the collection of finite sets is as the least collection containing the empty set and closed under the operation of adding one element to a set in the collection. There are many ways to represent this as a term.

One way is given in the TPS library by the polymorphic definition FINITE1 of type $(\sigma o)o$ which is defined to be the term

$$\lambda p.\forall w.w(\lambda x.\bot) \wedge (\forall rx.wr \rightarrow w(\mathsf{ADD1}\, r\, x)) \rightarrow wp$$

where p and r have type σo, w has type $(\sigma o)o$ and x has type σ. This definition is problematic if one works in weaker versions of simple type theory without extensionality. (Without extensionality, the same set may correspond to multiple characteristic functions.) Another definition in the TPS library which corresponds to finiteness (even in the absence of extensionality) is FINITE-SET-EXT of type $(\sigma o)o$ which is defined to be

$$\lambda X.\forall P.(\forall E.\neg(\exists t.Et) \rightarrow PE) \wedge (\forall YzZ.PY \wedge (\forall u.Zu \equiv \mathsf{ADD1}\, Y\, z\, u) \rightarrow PZ) \rightarrow PX$$

where E, X, Y, Z have type σo, P has type $(\sigma o)o$ and t and z have type σ. In words, FINITE-SET-EXT defines finiteness as the least collection of sets containing every empty set and every set that is equivalent to a set obtained by adjoining one element to a set in the collection. The definitions above are polymorphic in the sense that one can choose any type for σ. Let α be a fixed base type. In the TPS library, THM551 is simply stated as FINITE1 $C \rightarrow$ FINITE-SET-EXT C where C has type αo and FINITE1 and FINITE-SET-EXT are the α instances of the polymorphic definitions. Since the THF0 format of the TPTP does not support polymorphic definitions, we expand the α instances of the definitions of FINITE1, FINITE-SET-EXT and ADD1 to obtain the formula

$$\begin{gathered}(\forall w.w(\lambda x.\bot) \wedge (\forall rx.wr \rightarrow w(\lambda t.rt \vee t = x)) \rightarrow wC) \rightarrow \\ \forall P.(\forall E.\neg(\exists t.Et) \rightarrow PE) \wedge (\forall YzZ.PY \wedge (\forall u.Zu \equiv (Yu \vee u = z)) \rightarrow PZ) \rightarrow PC\end{gathered}$$

where w and P have type $(\alpha o)o$, r, C, E, Y and Z have type αo and x, t, z, u have type α. Up to the names of variables, this is the problem SEU868^5 from the TPTP. The variable names we use are the shorter TPS variable names (with one exception). If SPLIT_GLOBAL_DISJUNCTIONS is true, the initial branch contains

$$\begin{gathered}\forall w.w(\lambda x.\bot) \wedge (\forall rx.wr \rightarrow w(\lambda t.rt \vee t = x)) \rightarrow wC, \\ \forall E.(\forall t.\neg Et) \rightarrow PE, \\ \forall YzZ.PY \wedge (\forall u.Zu \equiv (Yu \vee u = z)) \rightarrow PZ, \\ \neg PC\end{gathered}$$

Several instantiations are used in the proof. Two subterms of this initial branch are P and $\lambda x.\bot$. Satallax uses P as an instantiation for w and $\lambda x.\bot$ as an instantiation for E if INITIAL_SUBTERMS_AS_INSTANTIATIONS is true. This yields new formulas

$$P(\lambda x.\bot) \wedge (\forall rx.Pr \rightarrow P(\lambda t.rt \vee t = x)) \rightarrow PC \text{ and } (\forall t.\neg\bot) \rightarrow P(\lambda x.\bot).$$

Processing the former formula leads to the introduction of the variables r and x. Because of mating, the terms r, x and $\lambda z.rz \vee z = x$ eventually appear as sides of disequations. This leads to the use of r as an instantiation for Y, x for z and finally $\lambda z.rz \vee z = x$ for Z if INSTANTIATE_WITH_FUNC_DISEQN_SIDES is true. This yields the formula $Pr \wedge (\forall u.(ru \vee u = x) \equiv (ru \vee u = x)) \rightarrow P(\lambda z.rz \vee z = x)$. Once these appropriate instantiations have been made, the rest of the search is easy.

At each stage of the search there are a number of options for continuing the search. We call these options *commands*. There are commands for processing active formulas and active instantiations. Executing such a command creates a successor state in which the formula or instantiation is passive. Whenever a command is generated (e.g., when a new active formula is created), the command is put into a priority queue. Priorities of different commands are computed using the values of various flags.

As mentioned earlier, in order to have a complete procedure, we need to enumerate closed terms to use as instantiations for function types. There are various commands that correspond to making progress in such an enumeration process. There are two (independent) enumeration procedures implemented in Satallax. The first enumeration procedure uses commands to request normal terms of a certain type in a certain context. The execution of each command corresponds to making a further commitment about the term requested. Once a term is completely determined, it is added as an active instantiation. The second enumeration procedure is used if the flag Enum_Iter_Deep is true. This alternate procedure periodically generates all closed terms using the current set of variables and logical constants up to a certain depth. For completeness, each time we enumerate again, we increase the depth. We next consider a simple example which demonstrates both enumeration procedures.

Example 4 The conjecture in the TPTP problem SYO548^1 is

$$\exists E.\forall P.(\exists X.\neg PX) \to \neg(P(EP))$$

where E has type $(\iota o)\iota$, P has type ιo and X has type ι. This states that there is an operator which maps each predicate P over a type ι to an element of ι that does not satisfy P, if such an element exists. The idea is to obtain such an operator by applying ε_ι to the complement of P. That is, we should prove this by instantiating with $(\lambda p.\varepsilon_\iota(\lambda x.\neg px))$ for E (where p has type ιo and x has type ι). Several modes can solve this problem in less than a second. We describe the search space generated by two modes: Mode81 (using the first enumeration procedure – with Enum_Iter_Deep false) and Mode261 (using the second enumeration procedure – with Enum_Iter_Deep true). Since $(\iota o)\iota$ is a quantified type, both enumeration procedures will attempt to generate instantiation terms of this type.

With Mode81 Satallax executes a sequence of commands including the following. Request a term of type $(\iota o)\iota$. After imitating ε_ι, request a term of the form $\lambda p.\varepsilon_\iota[]$ where p has type ιo and the hole $[]$ is yet to be determined. To fill the hole, request a term of type ιo where p may occur free. After imitating negation, request a term of the form $\lambda x.\neg[]$ where x has type ι and the hole $[]$ is yet to be determined. To fill the hole, request a term of type o where p and x may occur free. After projecting p, request a term of the form $p[]$ where $[]$ is yet to be determined. To fill this last hole, request a term of type ι in which p and x may occur free. Projecting x fills this hole and the completed term $(\lambda p.\varepsilon(\lambda x.\neg px))$ is added as an active instantiation. After this, the state quickly becomes propositionally unsatisfiable. No other potential instantiation of type $(\iota o)\iota$ is generated before this useful one. For this reason, by the time the state becomes propositionally unsatisfiable, only 15 clauses have been generated, 6 of which form an unsatisfiable core. Note that each request above is a separate command which is put into the priority queue and is hence interleaved with the other commands of the search procedure.

With Mode261 Satallax periodically enumerates terms of type $(\iota o)\iota$ up to a certain depth. The procedure generates terms in η-long form. That is, if the term should have type $\sigma_1 \cdots \sigma_n \alpha$ (where α is ι or o), then we assume it has the form

$$\lambda x^1 \cdots \lambda x^n.cs^1 \cdots s^m$$

where c is either a logical constant, a variable other than $x^1, \ldots, x^n$ or one of the variables $x^1, \ldots, x^n$. The depth of the term

$$\lambda x^1 \cdots \lambda x^n.cs^1 \cdots s^m$$

is the maximum depth of the subterms $s^1, \ldots, s^m$ plus 1 plus the depth value of c. The depth value of c is determined by the values of certain integer flags. For example, if $c \in \{x^1, \ldots, x^n\}$, then the depth value is determined by the value of the flag PROJECT_DELAY. With Mode261, the value of PROJECT_DELAY is 0. Similarly, values of flags set by Mode261 determine that $=_\iota$, $=_o$ and $\bot$ have depth value 0 and that ε_ι has depth value 100. Since the flag ENUM_ITER_DEEP_INIT has value 3, the first time the enumeration procedure is called it is limited to generating terms of depth at most 3. Such a term would be of the form $\lambda p.cs^1 \cdots s^m$ where p has type ιo. The only possible c would be of the form ε_σ, but each such c has depth value of at least 100. For this reason, no instantiations are generated. Each succeeding time the enumeration procedure is called, the depth limit is increased by 1 plus the value of the flag ENUM_ITER_DEEP_INCR. Since the value of the flag ENUM_ITER_DEEP_INCR is 1, each time the enumeration procedure is called it will increase the allowed depth by 2. Finally when the allowed depth is 103, the procedure will generate the following four terms:

$$\lambda p.\varepsilon_\iota(\lambda x.px),\ \lambda p.\varepsilon_\iota(\lambda x.x =_\iota x),\ \lambda p.\varepsilon_\iota(\lambda x.\bot =_o \bot),\ \lambda p.\varepsilon_\iota(\lambda x.\bot)$$

where x has type ι. Note that the depth value of p and x are 0 while the depth value of ε_ι is 100. Hence the depth of $(\lambda x.px)$ is 1 and the depth of $\lambda p.\varepsilon_\iota(\lambda x.px)$ is 102. The next time when the allowed depth is 105,365 terms are enumerated to be used as instantiations. One of these terms is $(\lambda p.\varepsilon(\lambda x.\bot = px))$, which leads to a successful proof. By the time the state becomes propositionally unsatisfiable, 6,053 propositional clauses have been generated; 11 of these clauses form an unsatisfiable core.

The next example has only been proven using a mode in which Enum_Iter_Deep is true.

Example 5 The TPTP problem GRA027^1 is a problem in propositional type theory. It represents the fact that the Ramsey number $R(3, 3)$ is greater than 4. The type oo has exactly four elements. The conjecture states that there is a symmetric relation G (of type $(oo)(oo)o$) with no 3-cliques and no 3-anticliques. Satallax can solve this problem with mode Mode261. Enumerating all terms of type $(oo)(oo)o$ with depth bounded by 3 leads to 13 possible instantiations. Either of the two terms $\lambda X.\lambda Y.X\bot = Y\bot$ or $\lambda X.\lambda Y.Y\bot = X\bot$ provide a solution (where X and Y are variables of type oo).

One of the most useful extensions implemented in Satallax is, under certain flag settings, to generate higher-order clauses with higher-order literals to be matched against formulas as the formulas are processed. Such higher-order clauses are only used when every existential variable in the clause has a strict occurrence in some literal. (A strict occurrence is essentially a pattern occurrence which is not below another existential variable [17].) The following example illustrates the use of higher-order clauses.

Example 6 The problem SEU506ˆ2 of the TPTP is a simple set theory example. We assume a membership relation $\in$ of type $\iota\iota o$ and an empty set $\emptyset$ of type ι. We make use of variables x, X, Y and A of type ι and a variable p of type o. We will write $s \in t$ for $\in\, s\, t$ and $s \notin t$ for $\neg(\in\, s\, t)$. The problem states that if we have

$$\forall x.x \in \emptyset \rightarrow \forall p.p,$$

and

$$\forall XY.(\forall x.x \in X \rightarrow x \in Y) \rightarrow (\forall x.x \in Y \rightarrow x \in X) \rightarrow X = Y,$$

then we know

$$\forall A.(\forall x.x \notin A) \rightarrow A = \emptyset.$$

Over a hundred modes can solve this problem in less than 2 seconds, many of which do not use higher-order clauses. An example of a mode using higher-order clauses that solves the problem is MODE205, which solves it in less than 0.1 s. Since SPLIT_GLOBAL_DISJUNCTIONS is true, the initial branch contains four formulas:

$$A \neq \emptyset$$
$$\forall x.x \notin A$$
$$\forall XY.(\forall x.x \in X \rightarrow x \in Y) \rightarrow (\forall x.x \in Y \rightarrow x \in X) \rightarrow X = Y$$
$$\forall x.x \in \emptyset \rightarrow \forall p.p$$

As these formulas are processed the following higher-order clauses are created:

$$^?x \notin A$$
$$^?X = {}^?Y \mid \neg(\forall x.x \in {}^?X \rightarrow x \in {}^?Y) \mid \neg(\forall x.x \in {}^?Y \rightarrow x \in {}^?X)$$
$$^?p \mid {}^?z \notin \emptyset$$

Here $^?x$, $^?X$, $^?Y$, $^?z$ and $^?p$ are existential variables. Each time a passive formula is processed, Satallax attempts to match literals in higher-order clauses against the negation of the formula. Each time a ground instance of a higher-order clause is generated corresponding propositional clauses are created and new active formulas are added to the state. In this particular example, the literal $^?X = {}^?Y$ matches the formula $A = \emptyset$ determining a ground instance of the second higher-order clause. This leads to the consideration of the two formulas $\neg(\forall x.x \in A \rightarrow x \in \emptyset)$ and $\neg(\forall x.x \in \emptyset \rightarrow x \in A)$. Two corresponding witnesses a and b are created and eventually the formulas $a \in A$, $a \notin \emptyset$, $b \in \emptyset$ and $b \notin A$. The sole literal in the unit clause $^?x \notin A$ matches the negation of the formula $a \in A$. Likewise, the higher-order clause $^?p \mid {}^?z \notin \emptyset$ contributes to the search since $^?z \notin \emptyset$ matches $b \notin \emptyset$ and $^?p$ matches an appropriate formula. (Many instantiations for $^?p$ could be useful here. The one Satallax uses happens to be $b \in \emptyset \rightarrow b \in A$.)

6 Results and Further Examples

TPTP v5.3.0 contains 2,924 problems in THF0 format. Among these, 347 are known to be satisfiable. (Satallax 2.4 terminates on many of these problems, recognizing them as satisfiable.) For 2,001 of the remaining 2,577 problems (77.6 %), there is some mode that Satallax 2.4 can use to prove the theorem (or show the assumptions are unsatisfiable) within 30 s. A strategy schedule running 61 modes for 5 min can solve each of the 2,001 problems.

One reason for the success of Satallax is that it can solve some problems by brute force. SEV106^5 is a TPTP problem which codes a Ramsey-style theorem about graphs and cliques. In Example 5 we proved a lower bound for $R(3,3)$ by providing an appropriate graph with 4 nodes. In this example we must prove an upper bound for $R(3,3)$ by proving no such appropriate graph with 6 nodes exists. In particular, we assume there are at least six distinct individuals and that there is a symmetric relation (i.e., an undirected graph) on individuals. There must be three distinct individuals all of whom are related or all of whom are unrelated. Since we are assuming there are six distinct individuals, we quickly have six corresponding discriminating terms. Satallax uses all six of these (blindly) as instantiations for the existential quantifiers, leading to 6^3 instantiations. Using mode MODE1 Satallax generates over 8,000 propositional clauses which MiniSat can easily recognize as unsatisfiable. In most examples only a handful of the clauses are the cause of unsatisfiability. In this example 284 clauses are used to show unsatisfiability.

Two higher-order examples from the TPTP that Satallax can solve are SYO378^5 and SYO379^5. These examples were created in TPS to illustrate the concept of quantificational depth, discussed at the end of [1]. Let c and x be variables of type ι and define $d_0 := \lambda x.x = c$. Furthermore, let y be a variable of type ιo and define $d_1 := \lambda y.y = d_0 \wedge \exists x.yx$. Finally, let z be a variable of type $(\iota o)o$ and define $d_2 := \lambda z.z = d_1 \wedge \exists y.zy$. One of the examples is $\exists y.d_1 y$ and the other is $\exists z.d_2 z$. A high-level proof is simply to note that $d_0 c$, $d_1 d_0$ and $d_2 d_1$ are all provable. However, if we expand all definitions, then these instantiations are no longer so easy to see. Fortunately, if the flag INSTANTIATE_WITH_FUNC_DISEQN_SIDES is set to TRUE, then d_0 and d_1 will appear as the side of a disequation and Satallax will include them as instantiations early. Verifying the instantiations work is not difficult. There are modes that can solve these problems within a second.

We also discuss two particularly interesting examples: SEV429^1 and SEV430^1. the TPTP. In both examples we use variables f, g of type $\iota\iota$ and variables x, y of type ι. SEV429^1 means every injective function f has a left inverse g and SEV430^1 means every surjective function f has a right inverse g.

$$(\forall x \forall y. fx = fy \to x = y) \to \exists g.\forall x.(g(fx)) = x \qquad \text{SEV429\^{}1}$$
$$(\forall y.\exists x. fx = y) \to \exists g.\forall x.(f(gx)) = x \qquad \text{SEV430\^{}1}$$

In both SEV429^1 and SEV430^1 Satallax must enumerate potential instantiations of type $\iota\iota$ for g. Some of the instantiations (e.g., $\lambda x.x$, f and $\lambda x.f(fx)$) are unhelpful and only serve to make the search space large. In both cases the instantiation used in the refutation is $\lambda y.\varepsilon x.fx = y$. An equivalent instantiation, $\lambda y.\varepsilon x.y = fx$, is also generated. (While it seems likely that such an equivalent instantiation could be discarded without sacrificing completeness, there is no currently known metatheoretic result to justify this intuition.)

Satallax can prove SEV429^1 using mode MODE214 in about 10 s. In the process it generates 15 candidates for g and 115,609 propositional clauses. Only 10 of the clauses are needed. Satallax can prove SEV430^1 using mode MODE207 in under 2 s. In the process it generates 16 higher-order instantiations (candidates for g) and 19,806 propositional clauses. It turns out that only 6 of these clauses are required to determine propositional unsatisfiability.

Satallax can also prove SEV430^1 using mode MODE286 in a few milliseconds. The reason is the flags EXISTSTOCHOICE, HOUNIF1 and HOUNIFMATE1 are set to TRUE. (Each of these flags is new to version 2.4 of Satallax.) When the flag EXISTSTOCHOICE is set to TRUE, the problem is preprocessed by using the choice operator ε to (essentially) Skolemize the problem. In particular, each negative occurrence of a formula $\forall_\sigma s$ is replaced by the logically equivalent formula $[s\,(\varepsilon x.\neg sx)]$ (where x is fresh). In SEV430^1 the problem is transformed to give two assumptions:

$$\forall y. f(\varepsilon x. fx = y) = y$$

and

$$\forall g. f(g(\varepsilon y. f(gy) \neq y)) \neq (\varepsilon y. f(gy) \neq y).$$

Note that the first assumption essentially contains the appropriate instantiation of g as a subterm. When the flags HOUNIF1 and HOUNIFMATE1 are set to TRUE, Satallax will attempt to generate instantiations by creating existential variables, mating potentially complementary formulas and performing higher-order unification. Since higher-order unification is undecidable, Satallax only attempts to find unifiers up to some certain depth. In the case of MODE286, the depth bound is 16, the value of the flag HOUNIF1BOUND. Depth-limited higher-order unification is not unique to Satallax; it has also been implemented in TPS [2] and LEO-II [5]. In this particular example, existential variables ${}^?Y$ and ${}^?G$ will be created and a disagreement pair of the form

$$(f(\varepsilon x. fx = {}^?Y) = {}^?Y) \ =^? \ (f({}^?G(\varepsilon y. f({}^?Gy) \neq y)) = (\varepsilon y. f({}^?Gy) \neq y)).$$

Higher-order unification can solve for an appropriate instantiation for ${}^?G$ very quickly.

Satallax has competed in the higher-order (THF) division of the CASC System Competition [20, 21] in 2010, 2011 and 2012. The other higher-order theorem provers competing in CASC have been TPS [2], LEO-II [5] and Isabelle [15]. Automated search in TPS is based on expansion proofs while search in LEO-II is based on a resolution calculus. Both TPS and LEO-II make essential use of existential variables which are partially instantiated during search. LEO-II was the first higher-order prover to take a cooperative approach. LEO-II makes calls to a first-order theorem prover to determine if the current set of higher-order clauses maps to an unsatisfiable set of first-order clauses. Isabelle uses Sledgehammer [16] which calls first-order provers and SMT solvers.

Satallax 1.4 (the last version of Satallax coded in Common Lisp) competed in the higher-order division of CASC in 2010 [20] and was able to prove 120 out of 200 problems, coming in second to LEO-II 1.2 which proved 125 out of 200 problems. Satallax 2.1 competed in the higher-order division of CASC in 2011 [21] and came in first by proving 246 out of 300 problems. LEO-II 1.2.8 came in second proving 208 of 300 problems. Satallax 2.4 competed in the higher-order division of CASC in 2012 and came in second by proving 132 out of 200 problems. Isabelle came in first proving 135 out of 200 problems. A demonstration version of Isabelle (with Sledgehammer) which also called LEO-II 1.3.2 and Satallax 2.4 was able to prove 166 out of 200 problems.

In [4], problems from first-order modal logics are considered. Since first-order modal logics can be embedded in higher-order logic, this gives a framework in which one can compare the performance of higher-order provers like Satallax and LEO-II with specialized first-order modal provers. There are five variants of first-order modal logic considered in Section 5 of [4]. For one of these variants, Satallax performs the best. For the other four variants, the (connection based) first-order modal prover MleanCoP performs the best with Satallax reported as second best. Satallax and LEO-II are reported to have the "broadest coverage" on these problems.

There are also results reported in Section 6.7.4 of [6]. These results are based on 1876 goals generated from theories of the higher-order prover Isabelle. On these problems, Satallax does not perform as well as LEO-II or the various first-order provers and SMT solvers considered in [6]. For example, Satallax is successful on 15.6 % of the goals while LEO-II is successful on 27 % of the goals and the first-order provers and SMT solvers are successful on over 40 % of the goals.

A natural question is how much of the runtime is spent by MiniSat. On the 200 problems used in the higher-order division of the CASC competition in 2012, the percentage of runtime used by MiniSat ranged from 0.7 to 89.7 %. On average, the percentage used by MiniSat was approximately 24.6 %. Currently, MiniSat is asked to search for propositional unsatisfiability every time new clauses are sent to MiniSat. Another natural question is how these percentages would change if we asked MiniSat to search for propositional unsatisfiability less often. For example, suppose we only asked MiniSat to search for propositional unsatisfiability once in every hundred times. (This option is not implemented in the current released version of Satallax, but was implemented to compute the following percentages.) On the same 200 problems, the percentage of the runtime used by MiniSat ranged from 0.2 to 33.2 %, with an average of approximately 8 %. On the other hand, with this modification Satallax times out on 3 of the 200 problems that Satallax 2.4 was able to prove without the modification. Based on these results, it seems a future implementation of Satallax should include a flag indicating how often MiniSat should search for propositional unsatisfiability.

7 Related Work

Smullyan introduced the notion of abstract consistency in 1963 [18]. One of his applications of abstract consistency is to justify reducing first-order unsatisfiability of a set M to propositional unsatisfiability of an extended set $R \cup M$. The procedure described in this paper and implemented in Satallax was developed without Smullyan's application in mind. Nevertheless, one can consider the procedure to be both an elaboration of Smullyan's idea as well as an extension to the higher-order case.

A different instantiation-based method Inst-Gen is described in [13]. Inst-Gen generates ground instances of first-order clauses and searches by interacting with a SAT-solver. This method is implemented in the first-order prover iProver [13, 14]. Note that iProver is also coded in Objective Caml and uses MiniSat via a foreign function interface. Two differences between the Inst-Gen method and the method in this paper should be noted. First, Inst-Gen assumes the problem is in clausal normal form. We do not make this assumption. As is well known, a substitution into a higher-order clause may lead to the need for further clause normalization. Second, Inst-Gen assumes an appropriate ordering on closures (clauses with substitutions). This ordering leads to important restrictions on inferences that can significantly improve the performance of Inst-Gen. We do not make use of any such ordering. In fact, a straightforward attempt to find such an ordering for the higher-order case is doomed to failure. This can be briefly indicated by an example. Suppose we define a closure to be a pair $C \cdot \theta$ of an atomic formula C and a substitution θ. The basic condition of a closure ordering $\succ$ (see [13]) is that $C \cdot \sigma \succ D \cdot \tau$ whenever $C\sigma = D\tau$ and $C\theta = D$ for some "proper instantiator" θ. In the higher-order case, we would consider equality of normal forms instead of strict syntactic equality. Consider two atomic formulas $C := p(\lambda xy. fxy)$ and $D := p(\lambda yx. fxy)$ where p, f, x and y are variables of appropriate types. Consider the substitution $\theta p := \lambda fxy.p(\lambda yx. fxy)$. Clearly $C\theta$ is β-equivalent to D and $D\theta$ is β-equivalent to C. An appropriate ordering (assuming θ would be considered a "proper instantiator") would need to have $C \cdot \emptyset \succ D \cdot \emptyset \succ C \cdot \emptyset$ where $\emptyset$ plays the role of the identity substitution.

As discussed in the previous section, three other higher-order theorem provers are Tps [1, 2], LEO-II [5] and Isabelle [6, 15].

8 Conclusion

We have given an abstract description of a search procedure for higher-order theorem proving. The key idea is to start with a notion of abstract consistency which integrates a restriction on instantiations. We gave a notion of a state which consists of finite sets of formulas, instantiations and propositional clauses. The invariants in the definition of a state correspond to the abstract consistency conditions. We have given a successor relation on states. Any fair strategy for choosing successors (until the set of propositional clauses is unsatisfiable) will give a complete theorem prover.

We have also described the implementation of this procedure as a higher-order theorem prover Satallax. Satallax is still new and there is a lot of room for improvement and further research. For example, it would improve Satallax significantly if one integrated procedures for solving for set variables as described in [7, 8]. Also, at the moment Satallax is very weak on problems that involve equational reasoning. Ideas for integrating efficient equational reasoning with the larger search procedure are needed.

Acknowledgements Thanks to Andreas Teucke whose work on translating Satallax refutations to Coq proof terms led to the consideration of some of the examples in this paper. Thanks also to Gert Smolka for his support and stimulating conversations. Finally, thanks to the anonymous reviewers for many helpful comments and suggestions.

References

1. Andrews, P.B., Bishop, M., Brown, C.E.: System description: TPS: a theorem proving system for type theory. In: McAllester, D. (ed.) Proceedings of the 17th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, vol. 1831, pp. 164–169. Springer-Verlag, Pittsburgh, PA (2000)

2. Andrews, P.B., Brown, C.E.: TPS: a hybrid automatic-interactive system for developing proofs. Journal of Applied Logic **4**(4), 367–395 (2006)
3. Backes, J., Brown, C.E.: Analytic tableaux for higher-order logic with choice. In: Jürgen Giesl, R.H. (ed.) Proceedings of the 5th International Joint Conference Automated Reasoning, IJCAR 2010, LNCS/LNAI, vol. 6173, pp. 76–90. Springer, Edinburgh, 16–19 July 2010
4. Benzmüller, C., Otten, J., Raths, T.: Implementing and evaluating provers for first-order modal logics. In: Raedt, L.D., Bessière, C., Dubois, D., Doherty, P., Frasconi, P., Heintz, F., Lucas, P.J.F. (eds.) Frontiers in Artificial Intelligence and Applications, ECAI, vol. 242, pp. 163–168. IOS Press (2012)
5. Benzmüller, C., Paulson, L., Theiss, F., Fietzke, A.: `LEO-II`—a cooperative automatic theorem prover for classical higher-order logic. In: 4th International Joint Conference on Automated Reasoning (IJCAR'08), LNCS (LNAI), vol. 5195, pp. 162–170. Springer (2008)
6. Blanchette, J.C.: Automatic Proofs and Refutations for Higher-Order Logic. Ph.D. thesis, Department of Informatics, T.U. München (2012)
7. Bledsoe, W.W., Feng, G.: Set-Var. J. Autom. Reason. **11**, 293–314 (1993)
8. Brown, C.E.: Solving for set variables in higher-order theorem proving. In: Voronkov, A. (ed.) Proceedings of the 18th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, vol. 2392, pp. 408–422. Springer-Verlag, Copenhagen (2002)
9. Brown, C.E.: Reducing higher-order theorem proving to a sequence of SAT problems. In: Bjørner, N., Sofronie-Stockkermans, V. (eds.) CADE—the 23rd International Conference on Automated Deduction, LNCS/LNAI, vol. 6803, pp. 147–161. Springer (2011)
10. Brown, C.E., Smolka, G.: Analytic tableaux for simple type theory and its first-order fragment. LMCS **6**(2), 1-33 (2010)
11. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011). doi:10.1145/1995376.1995394
12. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing, Lecture Notes in Computer Science, vol. 2919, pp. 333–336. Springer, Berlin/Heidelberg (2004)
13. Korovin, K.: iProver—an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek G. (eds.) Proceedings of the 4th International Joint Conference on Automated Reasoning, (IJCAR 2008), Lecture Notes in Computer Science, vol. 5195, pp. 292–298. Springer (2008)
14. Korovin, K., Sticksel, C.: iprover-eq: an instantiation-based theorem prover with equality. In: Jürgen Giesl, R.H. (ed.) Proceedings of the 5th International Joint Conference Automated Reasoning, IJCAR 2010, LNCS/LNAI, vol. 6173, pp. 196–201. Springer, Edinburgh, 16–19 July 2010
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic. In: LNCS, vol. 2283. Springer (2002)
16. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Ternovska, E., Schulz, S. (eds.) IWIL-2010 (2010)
17. Pfenning, F., Schürmann, C.: Algorithms for equality and unification in the presence of notational definitions. In: Altenkirch, T., Naraschewski, W., Reus, B. (eds.) TYPES 1998, Lecture Notes in Computer Science, vol. 1657, pp. 179–193. Springer (1999)
18. Smullyan, R.M.: A unifying principle in quantification theory. Proc. Natl. Acad. Sci. U.S.A. **49**, 828–832 (1963)
19. Sutcliffe, G.: The TPTP problem library and associated infrastructure: the FOF and CNF Parts, v3.5.0. J. Autom. Reasoning **43**(4), 337–362 (2009)
20. Sutcliffe, G.: The 5th IJCAR Automated Theorem Proving System Competition—CASC-J5. AI Commun. **24**(1), 75–89 (2011)
21. Sutcliffe, G.: The CADE-23 automated theorem proving system competition—CASC-23. AI Commun. **25**(1), 49–63 (2012)
22. Sutcliffe, G., Benzmüller, C.: Automated reasoning in higher-order logic using the TPTP THF infrastructure. J. Form. Reasoning **3**(1), 1–27 (2010)