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

Chad E. Brown

Saarland University, Saarbrücken, Germany

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 some experimental results.

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 case. Second, there is the problem of finding the correct instantiations for quantifiers. The instantiation problem appears in the first-order case. 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 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 new higher-order theorem prover Satallax[1]. The first implementation of Satallax was in Steel Bank Common Lisp. This earlier version, Satallax 1.4, competed in the higher-order division of CASC in 2010 [10]. Satallax 1.4 was able to prove 120 out of 200 problems, coming in second to LEO-II [4] which proved 125 out of 200 problems. The latest version of Satallax, Satallax 2.0, is implemented in Objective Caml. The SAT-solver MiniSat [6] (coded in C++) is used to determine propositional unsatisfiability.

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: $\bot : o$, $\rightarrow$: ooo, $\forall_\sigma : (\sigma o)o$, $=_\sigma$: $\sigma\sigma o$, $* : \iota$ and $\varepsilon_\sigma : (\sigma o)\sigma$. 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 $\neg\bot$, $s \neq_\sigma t$ (or $s \neq t$) means $\neg(s =_\sigma t)$, $\forall x.s$ means $\forall_\sigma \lambda x.s$ and $\varepsilon x.s$ means $\varepsilon_\sigma \lambda x.s$. Binders have as large a scope as is consistent with given parenthesis. For example, in $\forall x.px \rightarrow qx$ the occurrence of x in qx is bound by the $\forall$. The set $\mathcal{V}t$ of *free variables of* t is defined as usual.

An *accessibility context* ($\mathcal{C}$) 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 $\mathcal{C}[s]$ 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 $\mathcal{C}$ such that $\mathcal{C}[s] \in A$.

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. (In the implementation, de Bruijn indices are used.) 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

[1] Satallax is available at `satallax.com`

as x, we write θ_s^x for the substitution that agrees everywhere with θ except $\theta_s^x x = s$. For each substitution θ let $\hat{\theta}$ be the usual extension of θ to all terms in a capture-avoiding manner.

A *frame* $\mathcal{D}$ is a typed collection of nonempty sets such that $\mathcal{D}_o = \{0, 1\}$ and $\mathcal{D}_{\sigma\tau}$ is a set of total functions from $\mathcal{D}_\sigma$ to $\mathcal{D}_\tau$. An *assignment* $\mathcal{I}$ into $\mathcal{D}$ is a mapping from variables and logical constants of type σ into $\mathcal{D}_\sigma$. An assignment $\mathcal{I}$ is *logical* if it interprets each logical constant to be an element satisfying the corresponding logical property. For example, if $\mathcal{I}$ is logical, then $\mathcal{I}\bot = 0$. An assignment $\mathcal{I}$ is an *interpretation* if it can be extended in the usual way to be a total function $\hat{\mathcal{I}}$ mapping each Λ_σ into $\mathcal{D}_\sigma$. A *Henkin model* $(\mathcal{D}, \mathcal{I})$ is a frame $\mathcal{D}$ and a logical interpretation $\mathcal{I}$ into $\mathcal{D}$. We say formula s is *satisfied* by a Henkin model $(\mathcal{D}, \mathcal{I})$ if $\hat{\mathcal{I}}s = 1$. 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 $\mathcal{U}_\sigma^A \subseteq \Lambda_\sigma$ as follows.

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

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

We call a finite set of normal formulas a *branch*. A cut-free tableau calculus for higher-order logic with extensionality is given in [5]. 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 restriction is shown complete for the first-order case in [5]. 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. A set Γ of branches is an *abstract consistency class* if it satisfies all the conditions in Figure 1. This definition differs slightly from the one in [3] because we are using $\rightarrow$ 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 all logical connectives except negation.

$\mathcal{C}_\bot$	$\bot$ is not in A.
$\mathcal{C}_\neg$	If $\neg s$ is in A, then s is not in A.
$\mathcal{C}_{\neq}$	$s \neq_\iota s$ is not in A.
$\mathcal{C}_\rightarrow$	If $s \rightarrow t$ is in A, then $A \cup \{\neg s\}$ or $A \cup \{t\}$ is in Γ.
$\mathcal{C}_{\neg\rightarrow}$	If $\neg(s \rightarrow t)$ is in A, then $A \cup \{s, \neg t\}$ is in Γ.
$\mathcal{C}_\forall$	If $\forall_\sigma s$ is in A, then $A \cup \{[st]\}$ is in Γ for every $t \in \mathcal{U}^A_\sigma$.
$\mathcal{C}_{\neg\forall}$	If $\neg\forall_\sigma s$ is in A, then $A \cup \{\neg[sx]\}$ is in Γ for some variable x.
$\mathcal{C}_{\text{MAT}}$	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 $i \in \{1, \ldots, n\}$.
$\mathcal{C}_{\text{DEC}}$	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 $i \in \{1, \ldots, n\}$.
$\mathcal{C}_{\text{CON}}$	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.
$\mathcal{C}_{\text{BQ}}$	If $s =_o t$ is in A, then either $A \cup \{s, t\}$ or $A \cup \{\neg s, \neg t\}$ is in Γ.
$\mathcal{C}_{\text{BE}}$	If $s \neq_o t$ is in A, then either $A \cup \{s, \neg t\}$ or $A \cup \{\neg s, t\}$ is in Γ.
$\mathcal{C}_{\text{FQ}}$	If $s =_{\sigma\tau} t$ is in A, then $A \cup \{[\forall x.sx =_\tau tx]\}$ is in Γ for some $x \in \mathcal{V}_\sigma \setminus (\mathcal{V}s \cup \mathcal{V}t)$.
$\mathcal{C}_{\text{FE}}$	If $s \neq_{\sigma\tau} t$ is in A, then $A \cup \{\neg[\forall x.sx =_\tau tx]\}$ is in Γ for some $x \in \mathcal{V}_\sigma \setminus (\mathcal{V}s \cup \mathcal{V}t)$.
$\mathcal{C}_\varepsilon$	If $\varepsilon_\sigma s$ is accessible in A, then either $A \cup \{[s(\varepsilon s)]\}$ is in Γ or there is some $x \in \mathcal{V}_\sigma \setminus \mathcal{V}s$ such that $A \cup \{[\forall x.\neg(sx)]\}$ is in Γ.

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

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* $\mathcal{C}$ if there is some literal $l \in \mathcal{C}$ such that $\Phi l = 1$. An assignment Φ *satisfies a set* $\mathcal{S}$ of clauses if Φ satisfies $\mathcal{C}$ for all $\mathcal{C} \in \mathcal{S}$.

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 $\mathcal{I}s = \mathcal{I}t$ in every Henkin model $(\mathcal{D}, \mathcal{I})$.

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 $(\mathcal{D}, \mathcal{I})$ such that $\Phi \lfloor s \rfloor = \hat{\mathcal{I}}s$ for all $s \in \Lambda_o$.

4 States and Successors

Definition 1. *A* quasi-state Σ *is a 5-tuple* $(\mathfrak{F}^\Sigma_p, \mathfrak{F}^\Sigma_a, \mathfrak{U}^\Sigma_p, \mathfrak{U}^\Sigma_a, \mathfrak{C}^\Sigma)$ *where* $\mathfrak{F}^\Sigma_p$ *and* $\mathfrak{F}^\Sigma_a$ *are finite sets of normal formulas,* $\mathfrak{U}^\Sigma_p$ *and* $\mathfrak{U}^\Sigma_a$ *are finite sets of normal* *terms, and* $\mathfrak{C}^{\Sigma}$ *is a finite set of clauses. We call formulas in* $\mathfrak{F}_p^{\Sigma}$ passive formulas, *formulas in* $\mathfrak{F}_a^{\Sigma}$ active formulas, *terms in* $\mathfrak{U}_p^{\Sigma}$ passive instantiations *and terms in* $\mathfrak{U}_a^{\Sigma}$ active instantiations.

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

$$\mathfrak{F}^{\Sigma} := \mathfrak{F}_p^{\Sigma} \cup \mathfrak{F}_a^{\Sigma} \qquad \mathfrak{U}^{\Sigma} := \mathfrak{U}_p^{\Sigma} \cup \mathfrak{U}_a^{\Sigma} \qquad \mathfrak{U}_{p,\sigma}^{\Sigma} := \mathfrak{U}_p^{\Sigma} \cap \Lambda_\sigma \qquad \mathfrak{U}_{a,\sigma}^{\Sigma} := \mathfrak{U}_a^{\Sigma} \cap \Lambda_\sigma$$

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 two simple examples. In these examples 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).

Example 1. Let $p, q : o$ be variables. 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 $\mathfrak{F}_p^{\Sigma_0} = \emptyset$, $\mathfrak{F}_a^{\Sigma_0} = \{p, \forall q.p \to q\}$, $\mathfrak{U}_p^{\Sigma_0} = \{\bot, \top\}$, $\mathfrak{U}_a^{\Sigma_0} = \emptyset$ and $\mathfrak{C}^{\Sigma_0}$ 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 $\mathfrak{U}_p^{\Sigma_0}$. This results in a state Σ_1 in which $\mathfrak{F}_p^{\Sigma_1} = \{\forall q.p \to q\}$, $\mathfrak{F}_a^{\Sigma_1} = \{p, p \to \bot, p \to \top\}$, $\mathfrak{U}_p^{\Sigma_1} = \mathfrak{U}_p^{\Sigma_0}$, $\mathfrak{U}_a^{\Sigma_1} = \mathfrak{U}_a^{\Sigma_0}$ and $\mathfrak{C}^{\Sigma_1}$ contains the two unit clauses from $\mathfrak{C}^{\Sigma_0}$ 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 $\mathfrak{C}^{\Sigma_1}$. This completes the refutation. The two states can be displayed as in Figure 2.

	$\mathfrak{F}_p$	$\mathfrak{F}_a$	$\mathfrak{U}_p$	$\mathfrak{U}_a$	$\mathfrak{C}$
Σ_0		p, $\forall q.p \to q$	$\bot$, $\top$		$\lfloor p \rfloor$
					$\lfloor \forall q.p \to q \rfloor$
Σ_1	$\forall q.p \to q$	~~$\forall q.p \to q$~~			$\overline{\lfloor \forall q.p \to q \rfloor} \sqcup \overline{\lfloor p \rfloor}$
		$p \to \bot$, $p \to \top$			$\overline{\lfloor \forall q.p \to q \rfloor} \sqcup \lfloor p \to \top \rfloor$

Fig. 2. States from Example 1

Example 2. Let $p : \iota o$ and $x : \iota$ be variables. 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 (corresponding to assuming the negation) to a state Σ_7 such that $\mathfrak{C}^{\Sigma_7}$ is propositionally unsatisfiable. The states Σ_i for $i \in \{0, \ldots, 7\}$ are indicated in Figure 3. In the first step we process $\neg\forall x.px \to p(\varepsilon p)$ by choosing a fresh variable $y : \iota$ and including the new formula $\neg(py \to p(\varepsilon p))$ and a clause relating the literals corresponding to the two formulas. The resulting state is Σ_1. 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. We obtain Σ_3 by processing py. In general, processing such a formula involves mating it with all passive formulas of the form $\neg pt$. 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 $\neg p(\varepsilon p)$. 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. Also, since εp is accessible in $\mathfrak{F}_p^{\Sigma_5}$, 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. We obtain Σ_6 by processing $\forall x.\neg px$. In principle, this means instantiating with all passive instantiations of type ι, 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 $\forall x.\neg px$. 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.

	$\mathfrak{F}_p$	$\mathfrak{F}_a$	$\mathfrak{U}_p$	$\mathfrak{U}_a$	$\mathfrak{C}$
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. 3. States from Example 2

Definition 2. *A quasi-state $\Sigma = (\mathfrak{F}_p^\Sigma, \mathfrak{F}_a^\Sigma, \mathfrak{U}_p^\Sigma, \mathfrak{U}_a^\Sigma, \mathfrak{C}^\Sigma)$ is a* state *if the conditions in Figure 4 hold and for every clause $\mathcal{C}$ in $\mathfrak{C}^\Sigma$ and every literal $l \in \mathcal{C}$, either $l = \lfloor s\rfloor$ for some $s \in \mathfrak{F}^\Sigma$ or $l = \overline{\lfloor s\rfloor}$ for some $s \in \mathfrak{F}_p^\Sigma$.*

$\mathcal{S}_\bot$ If $\bot$ is in $\mathfrak{F}_p$, then $\overline{\lfloor\bot\rfloor}$ is in $\mathfrak{C}$.

$\mathcal{S}_{\neq}$ If $s \neq_\iota s$ is in $\mathfrak{F}_p$, then $\lfloor s = s\rfloor$ is in $\mathfrak{C}$.

$\mathcal{S}_{\to}$ If $s \to t$ is in $\mathfrak{F}_p$ and t is not $\bot$, then $\{\neg s, t\} \subseteq \mathfrak{F}$ and $\overline{\lfloor s \to t\rfloor} \sqcup \lfloor\neg s\rfloor \sqcup \lfloor t\rfloor$ is in $\mathfrak{C}$.

$\mathcal{S}_{\neg\to}$ If $\neg(s \to t)$ is in $\mathfrak{F}_p$, then $\{s, \neg t\} \subseteq \mathfrak{F}$, $\lfloor s \to t\rfloor \sqcup \lfloor s\rfloor$ and $\lfloor s \to t\rfloor \sqcup \lfloor\neg t\rfloor$ are in $\mathfrak{C}$.

$\mathcal{S}_\forall$ If $\forall_\sigma s$ is in $\mathfrak{F}_p$ and $t \in \mathfrak{U}_{p,\sigma}$, then $[st] \in \mathfrak{F}$ and $\overline{\lfloor\forall_\sigma s\rfloor} \sqcup \lfloor st\rfloor$ is in $\mathfrak{C}$.

$\mathcal{S}_{\neg\forall}$ If $\neg\forall_\sigma s$ is in $\mathfrak{F}_p$, then there is some variable x of type σ such that $\neg[sx] \in \mathfrak{F}$ and $\lfloor\forall_\sigma s\rfloor \sqcup \overline{\lfloor sx\rfloor}$ is in $\mathfrak{C}$.

$\mathcal{S}_{\text{MAT}}$ If $\delta s_1 \ldots s_n$ and $\neg\delta t_1 \ldots t_n$ are in $\mathfrak{F}_p$ where $n \geq 1$, then $s_i \neq t_i$ is in $\mathfrak{F}$ for each $i \in \{1, \ldots, n\}$ and $\overline{\lfloor\delta s_1 \ldots s_n\rfloor} \sqcup \lfloor\delta t_1 \ldots t_n\rfloor \sqcup \lfloor s_1 \neq t_1\rfloor \sqcup \cdots \sqcup \lfloor s_n \neq t_n\rfloor$ is in $\mathfrak{C}$.

$\mathcal{S}_{\text{DEC}}$ If $\delta s_1 \ldots s_n \neq_\iota \delta t_1 \ldots t_n$ is in $\mathfrak{F}_p$ where $n \geq 1$, then $s_i \neq t_i$ is in $\mathfrak{F}$ for each $i \in \{1, \ldots, n\}$ and $\lfloor\delta s_1 \ldots s_n = \delta t_1 \ldots t_n\rfloor \sqcup \lfloor s_1 \neq t_1\rfloor \sqcup \cdots \sqcup \lfloor s_n \neq t_n\rfloor$ is in $\mathfrak{C}$.

$\mathcal{S}_{\text{CON}}$ If $s =_\iota t$ and $u \neq_\iota v$ are in $\mathfrak{F}_p$, then $\{s \neq u, t \neq u, s \neq v, t \neq v\} \subseteq \mathfrak{F}$ and the following four clauses are in $\mathfrak{C}$:

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

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

$\mathcal{S}_{\text{BQ}}$ If $s =_o t$ is in $\mathfrak{F}_p$, then $\{s, t, \neg s, \neg t\} \subseteq \mathfrak{F}$ 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 $\mathfrak{C}$.

$\mathcal{S}_{\text{BE}}$ If $s \neq_o t$ is in $\mathfrak{F}_p$, then $\{s, t, \neg s, \neg t\} \subseteq \mathfrak{F}$ 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 $\mathfrak{C}$.

$\mathcal{S}_{\text{FQ}}$ If $s =_{\sigma\tau} t$ is in $\mathfrak{F}_p$, then there is some $x \in \mathcal{V}_\sigma \setminus (\mathcal{V}s \cup \mathcal{V}t)$ such that $[\forall x.sx =_\tau tx]$ is in $\mathfrak{F}$ and $\overline{\lfloor s = t\rfloor} \sqcup \lfloor\forall x.sx = tx\rfloor$ is in $\mathfrak{C}$.

$\mathcal{S}_{\text{FE}}$ If $s \neq_{\sigma\tau} t$ is in $\mathfrak{F}_p$, then there is some $x \in \mathcal{V}_\sigma \setminus (\mathcal{V}s \cup \mathcal{V}t)$ such that $[\neg\forall x.sx =_\tau tx]$ is in $\mathfrak{F}$ and $\lfloor s = t\rfloor \sqcup \lfloor\neg\forall x.sx = tx\rfloor$ is in $\mathfrak{C}$.

$\mathcal{S}_\varepsilon$ If $\varepsilon_\sigma s$ is accessible in $\mathfrak{F}_p$, then there is some $x \in \mathcal{V}_\sigma \setminus \mathcal{V}s$ such that $[s(\varepsilon s)]$ and $[\forall x.\neg(sx)]$ are in $\mathfrak{F}$ and $\lfloor s(\varepsilon s)\rfloor \sqcup [\forall x.\neg(sx)]$ is in $\mathfrak{C}$.

Fig. 4. Conditions on a quasi-state $\Sigma = (\mathfrak{F}_p, \mathfrak{F}_a, \mathfrak{U}_p, \mathfrak{U}_a, \mathfrak{C})$

We say a propositional assignment Φ *s*atisfies a state Σ if Φ satisfies $\mathfrak{C}^\Sigma$. 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 $\mathfrak{C}^\Sigma$. Note that checking whether Σ is propositionally satisfiable is simply a SAT-problem.

A variable x is *fresh* for a state Σ if x is not free in any $s \in \mathfrak{F}^\Sigma \cup \mathfrak{U}^\Sigma$.

Given a branch A, an *initial state Σ for A* is a state with $A \subseteq \mathfrak{F}^\Sigma$, and $\mathfrak{C}^\Sigma = \{\lfloor s\rfloor | s \in A\}$. (We require $A \subseteq \mathfrak{F}^\Sigma$ rather than $A \subseteq \mathfrak{F}^\Sigma_a$ 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 $\mathfrak{F}^\Sigma_p = \emptyset$, $\mathfrak{F}^\Sigma_a = A$, $\mathfrak{U}^\Sigma_p = \emptyset$, $\mathfrak{U}^\Sigma_a = \emptyset$ and $\mathfrak{C}^\Sigma = \{\lfloor s\rfloor | s \in A\}$.

Definition 3. *We say a state Σ' is a* successor *of a state Σ (and write $\Sigma \to \Sigma'$) if $\mathfrak{F}^\Sigma_p \subseteq \mathfrak{F}^{\Sigma'}_p$, $\mathfrak{F}^\Sigma_a \subseteq \mathfrak{F}^{\Sigma'}$, $\mathfrak{U}^\Sigma_p \subseteq \mathfrak{U}^{\Sigma'}_p$, $\mathfrak{U}^\Sigma_a \subseteq \mathfrak{U}^{\Sigma'}$, $\mathfrak{C}^\Sigma \subseteq \mathfrak{C}^{\Sigma'}$, 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 $\Sigma_A \rightarrow \Sigma'$, then A is unsatisfiable.*

Proof. Assume $(\mathcal{D}, \mathcal{I})$ is a Henkin model of A. Choose Φ such that $\Phi\lfloor s\rfloor = \hat{\mathcal{I}}s$ 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 $\Sigma_A \rightarrow \Sigma'$.

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 Figure 4 and (2) generate new active instantiations. To ensure soundness, when processing a formula $\neg\forall_\sigma s$ a procedure should choose a fresh variable x, add $\neg[sx]$ to $\mathfrak{F}_a$ and add $\lfloor\forall_\sigma s\rfloor \sqcup \overline{\lfloor sx\rfloor}$ to $\mathfrak{C}$.

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 α-*path *(or, simply* path*) is an α-sequence $\overline{\Sigma} = (\Sigma_i)_{i<\alpha}$ of propositionally satisfiable states such that $\Sigma_i \rightarrow \Sigma_{i+1}$ 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 $\forall_\sigma s \in \mathfrak{F}^{\Sigma_i}$. Such a path is* fair *if the following conditions hold:*

1. *For all $i < \alpha$ and $s \in \mathfrak{F}_a^{\Sigma_i}$ there is some $j \in [i, \alpha)$ such that $s \in \mathfrak{F}_p^{\Sigma_j}$.*
2. *If σ is a quantified type, then for all $i < \alpha$, $A \subseteq \mathfrak{F}^{\Sigma_i}$ and $t \in \mathcal{U}_\sigma^A$ there is some $j \in [i, \alpha)$ such that $t \in \mathfrak{U}_p^{\Sigma_j}$.*

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 Theorem 2 given 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 $A \subseteq \mathfrak{F}^{\Sigma_i}$ 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 $A \subseteq \mathfrak{F}^{\Sigma_i}$, then A is i-supported.*

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

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 an j-consistent branch. Let $A_1, \ldots, A_n$ be branches such that $A \subseteq A_l \subseteq \mathfrak{F}^{\Sigma_j}$ 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 $k \in [j, \alpha)$ 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 $k_1, \cdots, k_n \in [j, \alpha)$ 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.

- $\mathcal{C}_\bot$ Suppose $\bot \in A$ and A is i-consistent. By fairness there is some $j \in [i, \alpha)$ such that $\bot \in \mathfrak{F}_p^{\Sigma_j}$. By $\mathcal{S}_\bot$ the unit clause $\overline{\lfloor \bot \rfloor}$ is in $\mathfrak{C}^{\Sigma_j}$. This contradicts A being j-supported.
- $\mathcal{C}_\neg$ 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.
- $\mathcal{C}_\to$ Suppose $s \to t$ 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 $A \subseteq \mathfrak{F}^{\Sigma_i}$ and so $s \to t \in \mathfrak{F}^{\Sigma_i}$. By fairness there is some $j \in [i, \alpha)$ such that $s \to t \in \mathfrak{F}_p^{\Sigma_j}$. By $\mathcal{S}_\to$ we know $\{\neg s, t\} \subseteq \mathfrak{F}^{\Sigma_j}$ and $\overline{\lfloor s \to t \rfloor} \sqcup \overline{\lfloor s \rfloor} \sqcup \lfloor t \rfloor$ is in $\mathfrak{C}^{\Sigma_j}$. Note that $A \cup \{\neg s\} \subseteq \mathfrak{F}^{\Sigma_k}$ and $A \cup \{t\} \subseteq \mathfrak{F}^{\Sigma_k}$ for every $k \in [j, \alpha)$. Assume neither $A \cup \{\neg s\}$ nor $A \cup \{t\}$ is j-consistent. By Lemma 2 there is some $k \in [j, \alpha)$ 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 $\overline{\lfloor s \to t \rfloor} \sqcup \overline{\lfloor s \rfloor} \sqcup \lfloor t \rfloor$ is in $\mathfrak{C}^{\Sigma_k}$ and $\Phi\lfloor s \to t \rfloor = 1$, 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.
- $\mathcal{C}_{\neg\to}$ Suppose $\neg(s \to t)$ is in an i-consistent branch A. Since A is i-consistent, we know $\neg(s \to t) \in \mathfrak{F}^{\Sigma_i}$. By fairness there is some $j \in [i, \alpha)$ such that $\neg(s \to t) \in \mathfrak{F}_p^{\Sigma_j}$. By $\mathcal{S}_{\neg\to}$ we know $\{s, \neg t\} \subseteq \mathfrak{F}^{\Sigma_j}$, and both $\lfloor s \to t \rfloor \sqcup \lfloor s \rfloor$ and $\lfloor s \to t \rfloor \sqcup \overline{\lfloor t \rfloor}$ are in $\mathfrak{C}^{\Sigma_j}$. We prove $A \cup \{s, \neg t\}$ is j-consistent. Let $k \in [j, \alpha)$ be given. Since A is i-consistent, it has some pseudo-model Φ satisfying Σ_k. Since $\Phi\lfloor \neg(s \to t) \rfloor = 1$, 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.
- $\mathcal{C}_\forall$ Let A be an i-consistent branch such that $\forall_\sigma s \in A$ and $t \in \mathcal{U}_\sigma^A$. Note that $\forall_\sigma s \in A \subseteq \mathfrak{F}^{\Sigma_i}$ witnesses that σ is a quantified type on the path. By fairness there is some $j \in [i, \alpha)$ such that $\forall s \in \mathfrak{F}_p^{\Sigma_j}$ and $t \in \mathfrak{U}_p^{\Sigma_j}$. By $\mathcal{S}_\forall$ $[st] \in \mathfrak{F}^{\Sigma_j}$ and $\overline{\lfloor \forall_\sigma s \rfloor} \sqcup \lfloor st \rfloor$ is in $\mathfrak{C}^{\Sigma_j}$. We prove A is j-consistent. Let $k \in [j, \alpha)$ 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 $\mathfrak{C}^{\Sigma_j}$, 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 $\lfloor\cdot\rfloor$.)

$\mathcal{C}_{\neg\forall}$ 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 $\neg\forall s \in \mathfrak{F}_p^{\Sigma_j}$. By $\mathcal{S}_{\neg\forall}$ there is some variable x such that $\neg[sx] \in \mathfrak{F}^{\Sigma_j}$ and $\lfloor\forall_\sigma s\rfloor \sqcup \overline{\lfloor sx\rfloor}$ is in $\mathfrak{C}^{\Sigma_j}$. Let $k \in [j, \alpha)$ 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.

$\mathcal{C}_{\text{CON}}$ 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 $\mathfrak{F}_p^{\Sigma_j}$. By $\mathcal{S}_{\text{CON}}$ $\{s \neq u, t \neq u, s \neq v, t \neq v\} \subseteq \mathfrak{F}^{\Sigma_j}$ and the following four clauses are in $\mathfrak{C}^{\Sigma_j}$:

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

Assume neither $A \cup \{s \neq u, t \neq u\}$ nor $A \cup \{s \neq v, t \neq v\}$ is j-consistent. By Lemma 2 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 $A_0 \subseteq \mathfrak{F}^{\Sigma_0} \subseteq \mathfrak{F}^{\Sigma_j}$. Let Φ satisfy Σ_j. For each $s \in A_0$, the unit clause $\lfloor s\rfloor$ is in $\mathfrak{C}^{\Sigma_j}$ 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 [11]. Such a 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 a few hundred modes 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. This is an especially good idea if, for example, the conjecture is a conjunction. It 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 $\bot$ and $\neg\bot$ 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.

At each stage of the search there are a number of options for continuing the search. An example of an option is processing a particular active formula. Another option might be to work on enumerating instantiations of a given type. The different search options are put into a priority queue as they are generated. (The priority queue is modified to ensure every option is eventually considered.) Many flags control the priority given to different options.

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.

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. This is the only time Satallax uses existential variables. 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 [8].) We also allow for equational literals which can be used to perform some equational inference. Rather than give a full description of this extension, we give one example. Suppose we process a formula $\forall f \forall x \forall y.mf(cxy) = c(fx)(mfy)$ where $m : (\iota\iota)\iota\iota$, $c : \iota\iota\iota$, $f : \iota\iota$, and $x, y : \iota$. In addition to processing this in the usual way (applying all passive instantations of type $\iota\iota$), we can create a higher-order unit clause $mF(cXY) = c(FX)(mFY)$ where F, X and Y are existential variables. The first and last occurrences of F are strict. The first occurrence of X is strict. Both occurrences of Y are strict. Now, when processing a new formula s, Satallax uses higher-order pattern matching to check if s is of the form $C[mt(cuv)]$ for some t, u and v. If so, a propositional clause

$$\overline{\lfloor \forall f \forall x \forall y.mf(cxy) = c(fx)(mfy) \rfloor} \sqcup \overline{\lfloor C[mt(cuv)] \rfloor} \sqcup \lfloor C[c(tu)(mtv)] \rfloor$$

is added to the set of clauses and the formula $[C[c(tu)(mtv)]]$ is added to the set of active formulas to be processed later.

6 Results and Examples

TPTP v5.1.0 contains 2798 problems in THF0 format. Among these, 343 are known to be satisfiable. (Satallax 2.0 terminates on many of these problems, recognizing them as satisfiable.) For 1790 of the remaining 2455 problems (73%), there is some mode that Satallax 2.0 can use to prove the theorem (or show the assumptions are unsatisfiable) within a minute. For one other problem there is a mode that proves the theorem in 96 seconds. A strategy schedule running 36 modes for just over 10 minutes can solve each of the 1791 problems.

One reason for the success of Satallax is that it can solve some problems by brute force. An example of this is the first-order theorem SEV106ˆ5 from the TPTP. This is a Ramsey-style theorem about graphs and cliques. 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 8000 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 a 284 clauses are used to show unsatisfiability.

Two higher-order examples from the TPTP that Satallax can solve are SYO378ˆ5 and SYO379ˆ5. These examples were created in TPS to illustrate the concept of quantificational depth, discussed at the end of [1]. Let $c : \iota$ be a variable and define $d_0 := \lambda x : \iota.x = c$, $d_1 := \lambda y : \iota o.y = d_0 \wedge \exists x.yx$ and $d_2 := \lambda z : (\iota o)o.z = d_1 \wedge \exists y.zy$ (where $s \wedge t$ means $\neg(s \to \neg t)$ and $\exists x.s$ means $\neg \forall x.\neg s$). 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_0c, d_1d_0 and d_2d_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 that are not yet in the TPTP. In both examples we use variables $f, g : \iota\iota$ and $x, y : \iota$.

$$(\forall y.\exists x.fx = y) \to \exists g.\forall x.(f(gx)) = x \tag{1}$$

Formula (1) means every surjective function f has a right inverse g.

$$(\forall x\forall y.fx = fy \to x = y) \to \exists g.\forall x.(g(fx)) = x \tag{2}$$

Formula (2) means every injective function f has a left inverse g.

In both examples (1) and (2) 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 meta-theoretic result to justify this intuition.)

Satallax can prove (1) using mode MODE219 in under 6 seconds. In the process it generates 29 higher-order instantiations (candidates for g) and 17776 propositional clauses. It turns out that only 6 of these clauses are required to determine propositional unsatisfiability. Satallax can prove (2) using mode MODE218 in about a minute. In the process it generates 24 candidates for g and 117650 propositional clauses. Only 10 of the clauses are needed.

7 Related Work

Smullyan introduced the notion of abstract consistency in 1963 [9]. One of Smullyan's 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 [7]. 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 [7]. 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 [7]) 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 identity substitution.

Regarding higher-order theorem provers, two well-known examples are TPS [2] and LEO-II [4]. 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 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.

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. A version of Satallax last year proved to be competitive in the higher-order division of CASC in 2010 [10]. The latest implementation (a complete reimplementation in Objective Caml) is more closely integrated with the SAT-solver MiniSat [6]. The new implementation will compete in the higher-order division of CASC in 2011.

Satallax is still new and there is a lot of room for improvement and further research. One of the areas where much more research is needed involves generating *useful* higher-order instantiations.

Acknowledgements. Thanks to Andreas Teucke whose ongoing work on translating Satallax refutations to Coq proof terms has led to the consideration of some interesting examples, including the Ramsey-style theorem described in Section 6. Thanks also to Gert Smolka for his support and stimulating conversations.

References

1. Andrews, P.B., Bishop, M., Brown, C.E.: System description: TPS: A theorem proving system for type theory. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 164–169. Springer, Heidelberg (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: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 76–90. Springer, Heidelberg (2010)
4. Benzmüller, C.E., Paulson, L.C., Theiss, F., Fietzke, A.: LEO-II - A cooperative automatic theorem prover for classical higher-order logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 162–170. Springer, Heidelberg (2008)
5. Brown, C.E., Smolka, G.: Analytic tableaux for simple type theory and its first-order fragment. Logical Methods in Computer Science 6(2) (June 2010)
6. Eén, N., Sörensson, N.: An extensible sat-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 333–336. Springer, Heidelberg (2004)
7. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
8. 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. LNCS, vol. 1657, pp. 179–193. Springer, Heidelberg (1999)
9. Smullyan, R.M.: A unifying principle in quantification theory. Proceedings of the National Academy of Sciences, U.S.A 49, 828–832 (1963)
10. Sutcliffe, G.: The 5th IJCAR Automated Theorem Proving System Competition - CASC-J5. AI Communications 24(1), 75–89 (2011)
11. Sutcliffe, G., Benzmüller, C.: Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. Journal of Formalized Reasoning 3(1), 1–27 (2010)