Satallax: An Automatic Higher-Order Prover

Chad E. Brown

Saarland University, Saarbrücken, Germany

Abstract. Satallax is an automatic higher-order theorem prover that generates propositional clauses encoding (ground) tableau rules and uses MiniSat to test for unsatisfiability. We describe the implementation, focusing on flags that control search and examples that illustrate how the search proceeds.

Keywords: higher-order logic, simple type theory, higher-order theorem proving

1 Introduction

Satallax is an automatic theorem prover for classical higher-order logic with extensionality and choice. The search proceeds by generating propositional clauses that simulate tableau rules. Once the set of propositional clauses is unsatisfiable, the original higher-order problem is solved. An abstract description of the search procedure is given in [6]. The corresponding tableau calculus is proven sound and complete relative to Henkin models in [1], and the search procedure is proven sound and complete in [6].

In this system description we discuss the implementation of the search procedure. A number of flags can be used to guide search. We discuss the most important of these flags and give example problems from TPTP v5.3.0 [10] to illustrate how these flags affect the behavior of Satallax. (From now on, we use TPTP to refer to TPTP v5.3.0.)

Satallax won the THF division of the CASC-23 competition at CADE-23 in 2011 [11]. Out of 300 problems with a 5 minute time limit, Satallax 2.1 solved 246. LEO-II 1.2.8 [3] came in second, solving 208 problems. Among the 300 problems, there were 15 problems that only Satallax could solve. Most of these 15 problems were related to a choice operator. Since Satallax is the only system that directly supports reaonsing with choice operators, it clearly has an advantage on such problems. By contrast, there were 18 problems LEO could solve but no other system could. Many of these involved first-order equational reasoning (e.g., group theory problems).

The first versions of Satallax (1.0-1.4) were coded in Steel-Bank Common Lisp during 2009-2010. Starting with version 2.0 in 2010, Satallax has been implemented in Objective Caml with the exception of some code implementing a foreign function interface to MiniSat functions. MiniSat [8] is implemented in C++. The latest version of Satallax is Satallax 2.3 (approximately 13,000 lines Objective Caml code and 100 lines of C++) which uses MiniSat 2.2.0 (approximately 2,000 lines of C++). Satallax is available at `satallax.com`.

2 Preliminaries

We will assume familiarity with simple type theory and only briefly review to make the notation clear. A more detailed presentation can be found in [1]. Simple types σ, τ are either base types (o, ι, α, β) or function types $\sigma\tau$ (for functions from σ to τ). The type o is the type of propositions. Terms (s, t) are either variables (x, y, z, ...), logical constants ($\bot$, $\to$, $\forall_\sigma$, $=_\sigma$ and ε_σ), applications st or λ-abstractions $\lambda x.s$. Variables have a corresponding type and we only consider well-typed terms. A term of type o is called a formula. We write s_t^x for the capture-avoiding substitution of t for x in s.

We use notation $\forall x_\sigma.s$ or $\forall x.s$ (where x has type σ) for $\forall_\sigma(\lambda x.s)$. We use infix notation $s \to t$ and $s =_\sigma t$ (or $s = t$) for $(\to s)t$ and $(=_\sigma s)t$, respectively. We write $\neg s$ for $s \to \bot$. We write stu for $(st)u$ except $\neg st$ means $\neg(st)$. Since the THF problems in the TPTP problem library make use of logical connectives such as $\vee$ and $\wedge$, we also use notation $s \vee t$ for $\neg s \to t$, $s \wedge t$ for $\neg(s \to \neg t)$, $s \leftrightarrow t$ for $s =_o t$, $\exists x.s$ for $\neg\forall x.\neg s$ and $\exists! x.s$ for $\exists x.s \wedge \forall y.s_y^x \to x = y$. The scope of the λ, $\forall$, $\exists$ and $\exists!$ binders is as far to the right as is consistent with parentheses. We also use the usual notation for quantifying over several variables. For example, $\forall xy.s$ means $\forall x.\forall y.s$ and $\exists xy.s$ means $\exists x.\exists y.s$.

A β-redex is of the form $(\lambda x.s)t$ and this redex reduces to s_t^x. An η-redex is of the form $(\lambda x.sx)$ where x is not free in s and this redex reduces to s. We also reduce terms $\neg\neg s$ to s. All typed terms have a normal form. Satallax normalizes eagerly.

A branch A is a finite set of normal formulas. Given a theorem proving problem, we take all the axioms of the problem and combine them with the negation of the conjecture (if a conjecture is given) to form a branch A. The goal is then to prove A is (Henkin-)unsatisfiable.

3 Basic Search Procedure and Implementation

We briefly describe the link between higher-order formulas and propositional literals. The general technique of using a propositional abstraction is standard and is used by SMT solvers (e.g., see [7]). Let Atom be a set of propositional atoms. A *literal* is an atom a or a negated atom $\overline{a}$. Let $\overline{\overline{a}}$ be a. Let $\lfloor - \rfloor$ be a function from formulas to propositional literals such that $\lfloor \neg s \rfloor$ is $\overline{\lfloor s \rfloor}$. (We assume if $\lfloor s \rfloor = \lfloor t \rfloor$, then s and t are equivalent up to renaming and normalization.) A clause is a finite set of literals. We write a clause $\{l_1, \ldots, l_n\}$ as $l_1 \sqcup \cdots \sqcup l_n$. (We use $\sqcup$ instead of $\vee$ to distinguish the propositional clause level from the higher-order formula level.)

A *quasi-state* Σ is determined by sets of passive and active formulas, sets of passive and active terms (to be used as instantiations) and a set of propositional clauses. An active formula or term is one that must still be processed, while a passive formula or term is one that has already been processed and can now

only be used as a side formula when processing a new active formula or term.[1] A *state* is a quasi-state satisfying a finite number of conditions that can be found in [6]. The idea of the conditions can be easily summarized as follows: For every (instance of) a tableau rule that can be formed using passive formulas and passive terms, there are corresponding propositional clauses in the state. Also, every literal l in a clause is either $\lfloor s \rfloor$ for some active or passive formula s or is $\overline{\lfloor s \rfloor}$ for some passive formula s.

Given a branch A to refute, we can start from any initial state for A. A state is *initial for* A if for every formula s in A, s is either active or passive in the state and the unit clause $\lfloor s \rfloor$ is a clause in the state.

On an abstract level, a state is transformed into a successor state by processing an active formula (making the formula passive), by processing an active term (making the term passive), or by the generation of a new active term (to use as an instantiation). The successor state may have new active formulas, new active terms and new clauses. In reality, the situation is a bit more complicated. First, there must be an enumeration scheme that creates new active terms for higher-order quantifiers (if the original problem contains higher-order quantifiers). Also, there are two tableau rules with more than one principal formula: mating and confrontation.

We describe the mating rule. Suppose we are processing an active formula $ps_1 \cdots s_n$ where p is a variable. In order to process $ps_1 \cdots s_n$ we should, for each passive formula $\neg pt_1 \cdots t_n$ (a *mate*), make each disequation $s_i \neq t_i$ an active formula in the new state and add a clause

$$\overline{\lfloor ps_1 \cdots s_n \rfloor} \sqcup \lfloor pt_1 \cdots t_n \rfloor \sqcup \overline{\lfloor s_1 = t_1 \rfloor} \sqcup \cdots \sqcup \overline{\lfloor s_n = t_n \rfloor}.$$

The way the implementation actually handles this case is to create a command for each pair of mates. When the command is executed, the disequations are added as active formulas and the corresponding clause is added to the new state. The confrontation rule is similar to the mating rule, but operates on an equation $s =_\alpha t$ and a disequation $u \neq_\alpha v$ at a base type α (other than o).

The particular behavior of the search depends on 33 boolean flags and 79 integer flags. We will describe a few of these flags in Section 4 and give examples illustrating how they affect search.

Active and passive terms of a state are used as instantiations for quantifiers. In the implementation, the initial state always starts with two passive terms $\bot$ and $\neg\bot$ which act as instantiations for type o. New active terms s and t of a base type α (other than o) appear during the search when a disequation $s \neq_\alpha t$ is processed. If no active term of a base type α appears, then eventually a default element must be inserted as an active term. This default element will either be a new variable of type α, the term $\varepsilon_\alpha(\lambda x.\bot)$, or some term of type α that has appeared during the search already. For the sake of completeness, new active terms of function types must be enumerated during the search. There are two

[1] A reviewer pointed out that in some of the literature on superposition-based theorem proving, the terms "active" and "passive" are used in the opposite way. We keep the current terminology to be consistent with [6].

different enumeration processes for such terms. Under some flag settings other active terms are inserted into the state. Since some logical constants (e.g., $=_\sigma$) depend on a type σ, there is also an enumeration process for generating types.

A *command* is one of the following:

1. Process a formula s. Unless s is already passive (meaning it has already been processed), make s passive and add new active formulas, active terms, proposition clauses, and commands.
2. Process a term t_σ as an instantiation. Unless t is already passive, make t passive and for each passive formula $\forall_\sigma s$ add the normal form u of st as a new active formula, add the command of processing u, and add the clause $\overline{\lfloor \forall_\sigma s \rfloor} \sqcup \lfloor u \rfloor$.
3. Apply an instance of the mating rule.
4. Apply an instance of the confrontation rule.
5. Create a default element of a base type α.
6. Work on enumerating a new type. Once a type σ has been generated by the type enumeration process, we can imitate (in the sense of higher-order unification) logical constants $=_\sigma$, $\forall_\sigma$ and ε_σ when enumerating instantiation terms.
7. Work on enumerating a term of a given type σ with local variables $x_1, \ldots, x_n$.
8. Given a term $t_{\sigma_1 \cdots \sigma_n \alpha}$, work on enumerating a term of type α of the form $ts_1 \cdots s_n$ with local variables $x_1, \ldots, x_n$.
9. Use iterative deepening to enumerate all closed terms of a type up to a certain depth. This is an alternative to the previous enumeration commands which we will not discuss further.
10. Filter out a passive formula s if the set of clauses implies $\overline{\lfloor s \rfloor}$. We will not discuss filtering further.

A collection of commands is put into a priority queue. The purpose of many of the integer flags is to determine the priority of new commands as they are generated. Search proceeds by taking one of the highest priority commands and processing it.

The search ends successfully when the set of propositional clauses is propositionally unsatisfiable.

Example 1. We discuss the simple problem SYO357^5 from the TPTP in detail to illustrate the search procedure. The source for this problem is [2]. The conjecture is $(\forall P_{\alpha o}.(a \vee \neg a) \wedge Pu \to (b \vee \neg b) \wedge Pv) \to \forall Q_{\alpha o}.Qu \to Qv$. Let s^1 be the negation of this conjecture. We start with an initial state with a single active formula s^1 and a single clause $\lfloor s^1 \rfloor$. We process this formula, making it passive, adding two new active formulas s^2: $\forall P_{\alpha o}.(a \vee \neg a) \wedge Pu \to (b \vee \neg b) \wedge Pv$ and s^3: $\neg \forall Q_{\alpha o}.Qu \to Qv$ and new clauses $\overline{\lfloor s^1 \rfloor} \sqcup \lfloor s^2 \rfloor$ and $\overline{\lfloor s^1 \rfloor} \sqcup \lfloor s^3 \rfloor$. We process s^2, but since there are no passive terms of type αo this adds no new active formulas or clauses. Since this is the first time a universal quantifier over type αo has been encountered, we add the command for enumerating a term of type αo. We process s^3, using Q as a fresh variable, adding the active formula s^4: $\neg(Qu \to Qv)$ and clause $\overline{\lfloor s^3 \rfloor} \sqcup \lfloor s^4 \rfloor$. We process s^4 adding active formulas Qu and $\neg Qv$ and

clauses $\overline{\lfloor s^4 \rfloor} \sqcup \lfloor Qu \rfloor$ and $\overline{\lfloor s^4 \rfloor} \sqcup \overline{\lfloor Qv \rfloor}$. At this point, one thing Satallax will do is to process Qu and then $\neg Qv$ which adds the command for mating these two formulas. This line of actions does not contribute to the solution. Instead, we return to the command for enumerating a term of type αo. The command is executed by choosing a fresh variable x of type α and adding a command for enumerating a term s of type o with x free. We execute this new command by finding all possible heads for a term of the form $\lambda x.-$. In the language of higher-order unification, these heads are either the result of a projection or of an imitation. No projection is possible (because α is not o). Possible imitations are $Q_{\alpha o}$ and logical constants of the form $\varepsilon_{\sigma_1 \cdots \sigma_n o}$. The instantiation we want has Q at the head. One new command is to enumerate a term of type o with Q at the head and with x (possibly) free. To do this we only need to enumerate a term of type α with x free (to use as the argument of Q). We obtain such a term by projecting the local variable x and obtain the closed term $\lambda x.Qx$. In normal form, the term is Q. We add this as a new active term Q and immediately process this Q. We use this term Q as an instantiation for the passive universally quantified formula s^2 giving the new active formula s^5: $(a \vee \neg a) \wedge Qu \to (b \vee \neg b) \wedge Qv$ and new clause $\overline{\lfloor s^2 \rfloor} \sqcup \lfloor s^5 \rfloor$. The rest of the search is straightforward. We process s^5 and then the formulas that arise from continuing to process the resulting formulas. This yields the following clauses which (combined with the clauses above) are propositionally unsatisfiable.

$$\overline{\lfloor s^5 \rfloor} \sqcup \overline{\lfloor (a \vee \neg a) \wedge Qu \rfloor} \sqcup \lfloor (b \vee \neg b) \wedge Qv \rfloor$$
$$\lfloor (a \vee \neg a) \wedge Qu \rfloor \sqcup \overline{\lfloor a \vee \neg a \rfloor} \sqcup \overline{\lfloor Qu \rfloor}$$
$$\lfloor a \vee \neg a \rfloor \sqcup \overline{\lfloor a \rfloor}$$
$$\lfloor a \vee \neg a \rfloor \sqcup \lfloor a \rfloor$$
$$\overline{\lfloor (b \vee \neg b) \wedge Qv \rfloor} \sqcup \lfloor b \vee \neg b \rfloor$$
$$\overline{\lfloor (b \vee \neg b) \wedge Qv \rfloor} \sqcup \lfloor Qv \rfloor$$

4 Flags and Examples

We now consider a few of the most important flags that affect search.

Some flags affect what happens before the search begins. If the boolean flag LEIBEQ_TO_PRIMEQ is true, then subterms of the form $\forall P_{\sigma o}.Ps \to Pt$ or $\forall P.\neg Ps \to \neg Pt$ (where P is free in neither s nor t) are rewritten to $s =_\sigma t$. Also, subterms of the forms $\forall R_{\sigma\sigma o}.(\forall x.Rxx) \to Rst$ or $\forall R_{\sigma\sigma o}.\neg Rst \to \neg\forall x.Rxx$ (where R is free in neither s nor t) are rewritten to $s = t$. This is often a good idea because dealing with equalities is usually easier than dealing with higher-order quantifiers. Two particular examples from the TPTP in which this is good idea are SEV288^5 $(\lambda x_\alpha.\lambda y_\alpha.\forall q_\alpha.qx \to qy) = (\lambda x.\lambda y.x = y)$ and SEV121^5 $(\lambda x_\iota.\lambda y.x = y) = (\lambda x.\lambda y.\forall p_{\iota\iota o}.(\forall z.pzz) \to pxy)$. In both cases, the problem becomes trivial after rewriting the quantified formulas into equations. An example in which this is a bad idea is SYO357^5 (Example 1) because the conjecture becomes $(\forall P_{\alpha o}.(a \vee \neg a) \wedge Pu \to (b \vee \neg b) \wedge Pv) \to u = v$. The instantiation needed for P is $(\lambda z_\alpha.u = z)$ which is more complicated than the instantiation Q used in Example 1.

Another flag that controls preprocessing is Split_Global_Disjunctions. If this flag is true, then the initial branch is split into several branches each of which is refuted independently. We next consider an example where this is a good idea.

Example 2. The formula $(\forall xy.x = y \to \phi x \leftrightarrow \psi x) \to (\exists! x.\phi x) \leftrightarrow \exists! x.\psi x$ is the conjecture of SEU550^2 from the TPTP. This can be split into two independent branches to refute:

$$\{(\forall xy.x = y \to \phi x \leftrightarrow \psi x), \phi x, (\forall y.\phi y \to x = y), (\forall x.\psi x \to \neg\forall y.\psi y \to x = y)\}$$
$$\{(\forall xy.x = y \to \phi x \leftrightarrow \psi x), \psi x, (\forall y.\psi y \to x = y), (\forall x.\phi x \to \neg\forall y.\phi y \to x = y)\}$$

Each of these subgoals is solved in the same way. This is an example in which we must first instantiate with a default element before we obtain disequations that give more helpful instantiation terms as active terms.

An example for which setting Split_Global_Disjunctions to true is a bad idea is SYO181^5 (a propositional encoding of McCarthy's Mutilated Checkerboard problem [9]) because the preprocessing would split it into over 2^{271} independent subgoals. On the other hand, if Split_Global_Disjunctions is false, then SYO181^5 is easy to solve.

If the flag Initial_Subterms_As_Instantiations is true, then we seed the initial state with active terms for each subterm of the initial branch. (This would simplify the proofs in Example 2 since x is a subterm, and hence we avoid the need to instantiate with a default element.) If the flag Instantiate_With_Func_Diseqn_Sides is true, then each time a functional disequation $s \neq_{\sigma\tau} t$ is processed the terms s and t are added as active terms.

Example 3. The conjecture of the problem SEU868^5 from the TPTP states that one characterization of $C_{\alpha o}$ being a finite set implies another characterization.

$$(\forall w_{(\alpha o)o}.w(\lambda x.\bot) \wedge \forall r_{\alpha o}x_\alpha.wr \to w(\lambda t.rt \vee t = x) \to wC) \to$$
$$\forall P_{(\alpha o)o}.(\forall E_{\alpha o}.\neg(\exists t.Et) \to PE) \wedge$$
$$(\forall Y_{\alpha o}z_\alpha Z_{\alpha o}.PY \wedge \forall u_\alpha.Zu \equiv (Yu \vee u = z) \to PZ) \to PC$$

If Split_Global_Disjunctions is true, the initial branch contains

$$\forall w_{(\alpha o)o}.w(\lambda x.\bot) \wedge \forall r_{\alpha o}x_\alpha.wr \to w(\lambda t.rt \vee t = x) \to wC,$$
$$\forall E_{\alpha o}.(\forall t.\neg Et) \to PE,$$
$$\forall Y_{\alpha o}z_\alpha Z_{\alpha o}.PY \wedge \forall u_\alpha.Zu \equiv (Yu \vee u = z) \to PZ,$$
$$\neg PC$$

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. Processing the formula $P(\lambda x.\bot) \wedge \forall r_{\alpha o}x_\alpha.Pr \to P(\lambda t.rt \vee t = x) \to PC$ leads to the introduction of the variables r and x. Because of mating, the terms r, x and $\lambda z_\alpha.\neg rz \to z = x$ eventually appear as sides of a disequation. This leads to the use of r as an instantiation for Y, x for z and finally $\lambda z_\alpha.\neg rz \to z = x$ for Z if Instantiate_With_Func_Diseqn_Sides is true.

One of the most successful additions to the basic search procedure is the use of higher-order clauses and pattern unification to find instantiations. If the flag ENABLE_PATTERN_CLAUSES is set to true, then processing universally quantified formulas may generate higher-order clauses with meta-variables. For example, an assumption $\forall xy.x \subseteq y \to y \subseteq x \to x =_\iota y$ will generate a clause of the form $?X \not\subseteq ?Y | ?Y \not\subseteq ?X | ?X =_\iota ?Y$. Afterwards, whenever a ground term $s \neq_\iota t$ is processed, the propositional clause

$$\overline{\lfloor \forall xy.x \subseteq y \to y \subseteq x \to x = y \rfloor} \sqcup \overline{\lfloor s \subseteq t \rfloor} \sqcup \overline{\lfloor t \subseteq s \rfloor} \sqcup \lfloor s = t \rfloor$$

is added and $s \not\subseteq t$ and $t \not\subseteq s$ are added as active formulas. An example is SEU506^2 in the TPTP.

If the boolean flag TREAT_CONJECTURE_AS_SPECIAL is true, then the conjecture (and subformulas of the conjecture) are processed before the other formulas. The integer flag AXIOM_DELAY determines how long the other formulas are delayed. The integer flag RELEVANCE_DELAY delays formulas longer if they do not have variables in common with the conjecture.

There are many more flags we will not discuss here. There are 279 modes (collections of flag settings) in Satallax 2.3. Given a five minute timeout, the strategy schedule (sequence of modes with a timeout) contains 37 modes. The TPTP (v5.3.0) contains 2924 THF (higher-order) problems. Of these, 343 are known to be satisfiable. Of the remaining 2581 THF problems, the strategy schedule can prove 1817 (70%) – including the examples above.

5 Conclusion and Future Work

In terms of CASC, Satallax has already proven to be a successful prover. However, there is much room for improvement. One possibility would be to integrate Satallax with an SMT solver [7, 4]. Another possibility would be to solve for set variables using the techniques described in [5]. Also, integrating Satallax with an interactive proof assistant would provide new ground upon judging its effectiveness. Preliminary steps have been taken to integrate Satallax with the proof assistant Coq.

References

1. Julian Backes and Chad E. Brown. Analytic Tableaux for Higher-Order Logic with Choice. *Journal of Automated Reasoning*, 47(4):451–479, 2011.
2. Christoph Benzmüller. *Equality and Extensionality in Automated Higher-Order Theorem Proving*. PhD thesis, Universität des Saarlandes, 1999.
3. Christoph Benzmüller, Frank Theiss, Larry Paulson, and Arnaud Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. In *Fourth International Joint Conference on Automated Reasoning (IJCAR'08)*, volume 5195 of *LNAI*. Springer, 2008.

4. Jasmin Christian Blanchette, Sasche Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT Solvers. In Nikolaj Bjørner and Viorica Sofronie-Stockkermans, editors, *CADE the 23rd International Conference on Automated Deduction*, LNCS/LNAI 6803, pages 116 – 130. Springer, Jul 2011.
5. Chad E. Brown. Solving for Set Variables in Higher-Order Theorem Proving. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 408–422, Copenhagen, Denmark, 2002. Springer-Verlag.
6. Chad E. Brown. Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems. In Nikolaj Bjørner and Viorica Sofronie-Stockkermans, editors, *CADE the 23rd International Conference on Automated Deduction*, LNCS/LNAI 6803, pages 147 – 161. Springer, Jul 2011.
7. Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.
8. Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer Berlin / Heidelberg, 2004.
9. John McCarthy. A Tough Nut for Proof Procedures, July 1964. Stanford Artificial Intelligence Memo No. 16.
10. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
11. G. Sutcliffe. The CADE-23 Automated Theorem Proving System Competition - CASC-23. *AI Communications*, page To appear, 2012.