An Abstract Framework for Satisfiability Modulo Theories

Cesare Tinelli

The University of Iowa

Credits

- ***Based on joint work with:***

  Clark Barrett, Peter Baumgartner, Robert Nieuwenhuis, and Albert Oliveras

- ***Special thanks to:***

  - the TABLEAUX 2007 PC for the invitation.

Satisfiability Modulo Theories

- Any SAT solver can be used to decide the satisfiability of ground (i.e., variable-free) first-order formulas.

Satisfiability Modulo Theories

- Any SAT solver can be used to decide the satisfiability of ground (i.e., variable-free) first-order formulas.
- Often, however, one is interested in the satisfiability of certain ground formulas in a theory:

Satisfiability Modulo Theories

- Any SAT solver can be used to decide the satisfiability of ground (i.e., variable-free) first-order formulas.
- Often, however, one is interested in the satisfiability of certain ground formulas in a theory:
  - Hardware verification: theory of equality, of bit vectors.

Satisfiability Modulo Theories

- Any SAT solver can be used to decide the satisfiability of ground (i.e., variable-free) first-order formulas.
- Often, however, one is interested in the satisfiability of certain ground formulas in a theory:
  - Hardware verification: theory of equality, of bit vectors.
  - Timed automata, planning: theory of integers/reals.

Satisfiability Modulo Theories

- Any SAT solver can be used to decide the satisfiability of ground (i.e., variable-free) first-order formulas.
- Often, however, one is interested in the satisfiability of certain ground formulas in a theory:
  - Hardware verification: theory of equality, of bit vectors.
  - Timed automata, planning: theory of integers/reals.
  - Software verification/model checking, compiler optimization: combinations of various theories.

Satisfiability Modulo Theories

- Any SAT solver can be used to decide the satisfiability of ground (i.e., variable-free) first-order formulas.
- Often, however, one is interested in the satisfiability of certain ground formulas in a theory:
  - Hardware verification: theory of equality, of bit vectors.
  - Timed automata, planning: theory of integers/reals.
  - Software verification/model checking, compiler optimization: combinations of various theories.
- We refer to this general problem as (ground) Satisfiability Modulo Theories, or SMT.

Satisfiability Modulo a Theory $\mathcal{T}$

Ground $\mathcal{T}$-satisfiability problem for a theory $\mathcal{T}$:

Is there a model of $\mathcal{T}$ that satisfies a given ground formula φ ?

Satisfiability Modulo a Theory $\mathcal{T}$

Ground $\mathcal{T}$-satisfiability problem for a theory $\mathcal{T}$:
Is there a model of $\mathcal{T}$ that satisfies a given ground formula φ ?

Some popular theories

- Equality with "Uninterpreted Functions"
- Arithmetic (Real and Integer)
- Arrays
- Bit vectors
- Sets
- Algebraic Datatypes (tuples, lits, etc.)

Satisfiability Modulo a Theory $\mathcal{T}$

- Note: The $\mathcal{T}$-satisfiability of ground formulas is decidable iff the $\mathcal{T}$-satisfiability of sets of literals is decidable

Satisfiability Modulo a Theory $\mathcal{T}$

- Note: The $\mathcal{T}$-satisfiability of ground formulas is decidable iff the $\mathcal{T}$-satisfiability of sets of literals is decidable
- Problem: In practice, dealing with Boolean combinations of literals is as hard as in the propositional case

Satisfiability Modulo a Theory $\mathcal{T}$

- Note: The $\mathcal{T}$-satisfiability of ground formulas is decidable iff the $\mathcal{T}$-satisfiability of sets of literals is decidable
- Problem: In practice, dealing with Boolean combinations of literals is as hard as in the propositional case
- Current solution: Exploit propositional satisfiability technology

Satisfiability Modulo a Theory $\mathcal{T}$

- Note: The $\mathcal{T}$-satisfiability of ground formulas is decidable iff the $\mathcal{T}$-satisfiability of sets of literals is decidable
- Problem: In practice, dealing with Boolean combinations of literals is as hard as in the propositional case
- Current solution: Exploit propositional satisfiability technology
- Favorite SAT technology: based on the Davis-Putnam-Loveland-Logemann (DPLL) procedure

Lifting SAT Technology to SMT

Lifting SAT Technology to SMT

- *Eager approach* [CBMC, UCLID, . . .]:
  - translate φ into an equisat. propositional formula,
  - feed it to any SAT solver.

Lifting SAT Technology to SMT

- *Eager approach* [CBMC, UCLID, . . .]:
  - translate φ into an equisat. propositional formula,
  - feed it to any SAT solver.
- *Lazy approach* [Barcelogic, CVC*, ICS, MathSAT, Verifun, Yices, Z3, . . .]:
  - treat φ as a propositional formula,
  - feed it to a DPLL-based SAT solver,
  - use a theory decision procedure to refine the formula,
  - use the decision procedure to guide the search of DPLL solver.

Lifting SAT Technology to SMT

- *Eager approach* [CBMC, UCLID, . . .]:
  - translate φ into an equisat. propositional formula,
  - feed it to any SAT solver.
- *Lazy approach* [Barcelogic, CVC*, ICS, MathSAT, Verifun, Yices, Z3, . . .]:
  - treat φ as a propositional formula,
  - feed it to a DPLL-based SAT solver,
  - use a theory decision procedure to refine the formula,
  - use the decision procedure to guide the search of DPLL solver.
- This talk focuses on the lazy approach.

An Abstract Framework for SMT

Lazy approach:

- treat φ as a propositional formula,
- feed it to a DPLL-based SAT solver,
- use a theory decision procedure to refine the formula,
- use the decision procedure to guide the search of DPLL solver.

There are several variants of this approach.

They can be modeled abstractly and declaratively as transition systems.

An Abstract Framework for SMT

Using transition systems helps:

- Skip over implementation details and unimportant control aspects.

An Abstract Framework for SMT

Using transition systems helps:

- Skip over implementation details and unimportant control aspects.
- Reason formally about DPLL-based solvers for SAT and for SMT.

An Abstract Framework for SMT

Using transition systems helps:

- Skip over implementation details and unimportant control aspects.
- Reason formally about DPLL-based solvers for SAT and for SMT.
- Model modern features such as non-chronological bactracking, lemma learning or restarts.

An Abstract Framework for SMT

Using transition systems helps:

- Skip over implementation details and unimportant control aspects.
- Reason formally about DPLL-based solvers for SAT and for SMT.
- Model modern features such as non-chronological bactracking, lemma learning or restarts.
- Describe different strategies and prove their correctness.

An Abstract Framework for SMT

Using transition systems helps:

- Skip over implementation details and unimportant control aspects.
- Reason formally about DPLL-based solvers for SAT and for SMT.
- Model modern features such as non-chronological bactracking, lemma learning or restarts.
- Describe different strategies and prove their correctness.
- Compare different systems at a higher level.

An Abstract Framework for SMT

Using transition systems helps:

- Skip over implementation details and unimportant control aspects.
- Reason formally about DPLL-based solvers for SAT and for SMT.
- Model modern features such as non-chronological bactracking, lemma learning or restarts.
- Describe different strategies and prove their correctness.
- Compare different systems at a higher level.
- Get new insights for further enhancements.

DPLL Procedure vs. Tableaux

***Grand* claim of the day:**

> Modern variants of DPLL can be understood as highly optimized proof procedures for the ground clause tableau calculus

DPLL Procedure vs. Tableaux

***Grand* claim of the day:**

Modern variants of DPLL can be understood as highly optimized proof procedures for the ground clause tableau calculus

Modeling clause tableaux too as transition systems helps see this connection

Clause Tableaux as Transitions Systems

States:

$$fail \quad \text{or} \quad T \parallel F$$

where $T = \{B_1, \ldots, B_k\}$ is a set of branches B_i
$B_i = (l_1, \ldots, l_{n_i})$ is a sequence of (ground) literals
$F = \{C_1, \ldots, C_p\}$ is a set of (ground) clauses.

Clause Tableaux as Transitions Systems

States:

$$\mathit{fail} \quad \text{or} \quad T \parallel F$$

Initial state:

- $\{\{\top\}\} \parallel F$ where F is to be checked for satisfiability

Expected final states:

- fail, if F is unsatisfiable
- $T \cup \{B\} \parallel G$ where G is logically equivalent to F and B satisfies G, if F is satisfiable

Clause Tableaux as Transitions Systems

States:

$$fail \quad \text{or} \quad T \parallel F$$

Notation:

- $T;\ B\,l \parallel F, C$ stands for $T \cup \{B \cdot (l)\} \parallel F \cup \{C\}$

Convention:

- We will treat consistent branches B as (partial) truth assignments

Transition Rules for a Basic Clause Tableau

Close

$T; B \parallel F \quad \rightarrow \quad T \parallel F$ **if** B is inconsistent (i.e., $p, \neg p \in B$)

Transition Rules for a Basic Clause Tableau

Close

$T;\ B \ \|\ F \quad \rightarrow \quad T \ \|\ F$ **if** B is inconsistent (i.e., $p, \neg p \in B$)

Expand

$$T;\ B \ \|\ F, l_1 \vee \cdots \vee l_n \quad \rightarrow \quad T;\ B\, l_1; \ldots; B\, l_n \ \|\ F, l_1 \vee \cdots \vee l_n \quad \textbf{if } (*)$$

$$(*) = \begin{cases} B \text{ is consistent} \\ B \not\models l_1 \vee \cdots \vee l_n \end{cases}$$

Transition Rules for a Basic Clause Tableau

Close

$T;\ B \parallel F \quad \longrightarrow \quad T \parallel F$ **if** B is inconsistent (i.e., $p, \neg p \in B$)

Expand

$$T;\ B \parallel F, l_1 \vee \cdots \vee l_n \quad \longrightarrow \quad T;\ B\,l_1;\ \ldots;\ B\,l_n \parallel F, l_1 \vee \cdots \vee l_n \quad \textbf{if } (*)$$

$$(*) = \begin{cases} B \text{ is consistent} \\ B \not\models l_1 \vee \cdots \vee l_n \end{cases}$$

Empty

$\emptyset \parallel F \quad \longrightarrow \quad \mathit{fail}$

The rules define a *transition relation* $\rightarrow$ over states.

Proof Procedures as Rule Application Strategies

- A *derivation (of a clause set F)* is a $\rightarrow$-chain starting with $\top \parallel F$.
- A finite derivation $\top \parallel F \rightarrow \cdots \rightarrow S$ is *exhausted* if S
  - is $T; B \parallel G$ where B is consistent and (propositionally) entails G ($B \models G$), or
  - is irreducible by the rules.
- A rule application strategy is *fair* if it stops only with an exhausted derivation.

Proof Procedures as Rule Application Strategies

Proposition Every fair rule application strategy for ground clause tableaux is:

- Terminating: it generates only finite derivations.
- Sound: it generates a derivation $\top \parallel F \to \cdots \to \mathit{fail}$ only if F is unsatisfiable.
- Complete: it can generate a derivation $\top \parallel F \to \cdots \to \mathit{fail}$ if F is unsatisfiable.
- Proof confluent: it can extend any derivation of $\top \parallel F$ with unsatisfiable F to one ending in fail.
- Model finding: it stops with state $\top \parallel F \to \cdots \to T \parallel G$ only if a branch of T is a model of F.

Enhancements to Basic Clause Tableaux

Additional rules

Conflict

$$T;\ B \parallel F, C \ \ \longrightarrow \ \ T \parallel F, C \quad \textbf{if } B \models \neg C$$

C is a *conflicting* clause

Enhancements to Basic Clause Tableaux

Additional rules

Conflict

$$T;\ B \parallel F, C \ \ \longrightarrow \ \ T \parallel F, C \quad \textbf{if } B \models \neg C$$

C is a *conflicting* clause

Propagate

$$T;\ B \parallel F, C \vee l \ \ \longrightarrow \ \ T;\ B\,l \parallel F, C \vee l \quad \textbf{if } \begin{cases} B \models \neg C \\ l \text{ is undefined in } B \end{cases}$$

Enhancements to Basic Clause Tableaux

Additional rules

Conflict

$$T;\ B \parallel F, C \quad\longrightarrow\quad T \parallel F, C \quad \textbf{if } B \models \neg C$$

C is a *conflicting* clause

Propagate

$$T;\ B \parallel F, C \vee l \quad\longrightarrow\quad T;\ B\,l \parallel F, C \vee l \quad \textbf{if } \begin{cases} B \models \neg C \\ l \text{ is undefined in } B \end{cases}$$

Split (atomic cut)

$$T;\ B \parallel F \quad\longrightarrow\quad T;\ B\,l;\ B\,\bar{l} \parallel F \quad \textbf{if } \begin{cases} l \text{ or } \bar{l} \text{ occurs in } F, \\ l \text{ is undefined in } B \end{cases}$$

Enhancements to Basic Clause Tableaux

Proposition Any fair strategy remains fair when restricted to use only **Split**, **Propagate**, **Conflict**, and **Fail**

Enhancements to Basic Clause Tableaux

Proposition Any fair strategy remains fair when restricted to use only **Split**, **Propagate**, **Conflict**, and **Fail**

Since these rules are branch local, we can build the tableau lazily, one branch at a time

Enhancements to Basic Clause Tableaux

Proposition Any fair strategy remains fair when restricted to use only **Split**, **Propagate**, **Conflict**, and **Fail**

Since these rules are branch local, we can build the tableau lazily, one branch at a time

Technically, we replace:

1. states $T \parallel F$ with states $B \parallel F$ where B is now a sequence of annotated literals
2. **Split** with **Decide**
3. **Conflict** with **Backtrack**
4. **Empty** with **Fail**

Enhancements to Basic Clause Tableaux

Proposition Any fair strategy remains fair when restricted to use only **Split**, **Propagate**, **Conflict**, and **Fail**

Since these rules are branch local, we can build the tableau lazily, one branch at a time

What we get at the end is a basic version of DPLL

Enhancements to Basic Clause Tableaux

Split

$$T;\, B \parallel F \;\longrightarrow\; T;\, B\, l;\, B\, \bar{l} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \bar{l} \text{ occurs in } F, \\ l \text{ is undefined in } B \end{cases}$$

becomes

Decide

$$B \parallel F \;\longrightarrow\; B\, l^{\bullet} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \bar{l} \text{ occurs in } F, \\ l \text{ is undefined in } B \end{cases}$$

Notation: $l^{\bullet}$ is l annotated as a *decision literal*

Enhancements to Basic Clause Tableaux

Conflict

$$T;\, B \parallel F, C \quad \longrightarrow \quad T \parallel F, C \quad \textbf{if } B \models \neg C$$

becomes

Backtrack

$$B_1\, l^\bullet\, B_2 \parallel F, C \quad \longrightarrow \quad B_1\, \bar{l} \parallel F, C \quad \textbf{if } \begin{cases} B_1\, l^\bullet\, B_2 \models \neg C, \\ l^\bullet \text{ rightmost dec. literal} \end{cases}$$

Enhancements to Basic Clause Tableaux

Empty

$$\emptyset \parallel F \quad \longrightarrow \quad \mathit{fail}$$

becomes

Fail

$$B \parallel F, C \quad \longrightarrow \quad \mathit{fail} \quad \textbf{if} \begin{cases} B \models \neg C, \\ B \text{ contains no decision literals} \end{cases}$$

Our Abstract Version of the Original DPLL

Propagate

$$B \parallel F, C \vee l \;\; \longrightarrow \;\; B, l \parallel F, C \vee l \quad \textbf{if} \begin{cases} B \models \neg C \\ l \text{ is undefined in } B \end{cases}$$

Decide $$B \parallel F \;\; \longrightarrow \;\; B\, l^{\bullet} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \bar{l} \text{ occurs in } F, \\ l \text{ is undefined in } B \end{cases}$$

Fail

$$B \parallel F, C \;\; \longrightarrow \;\; \mathit{fail} \quad \textbf{if} \begin{cases} B \models \neg C, \\ B \text{ contains no decision literals} \end{cases}$$

Backtrack

$$B_1\, l^{\bullet}\, B_2 \parallel F, C \;\; \longrightarrow \;\; B_1\, \bar{l} \parallel F, C \quad \textbf{if} \begin{cases} B_1\, l^{\bullet}\, B_2 \models \neg C, \\ l \text{ last decision literal} \end{cases}$$

Smarter Backtracking

Backtrack

$$B_1\, l^\bullet\, B_2 \;\|\; F, C \quad\to\quad B_1\, \bar{l} \;\|\; F, C \quad \textbf{if} \begin{cases} B_1\, l^\bullet\, B_2 \models \neg C, \\ l \text{ last decision literal} \end{cases}$$

is replaced in modern implementations by

Backjump

$$B_1\, l^\bullet\, B_2 \;\|\; F, C \quad\to\quad B_1\, k \;\|\; F, C \quad \textbf{if} \begin{cases} 1.\ B_1\, l^\bullet\, B_2 \models \neg C, \\ 2.\ \text{for some clause } D \vee k \\ \quad F, C \models D \vee k, \\ \quad B_1 \models \neg D, \\ \quad k \text{ is undefined in } B_1, \\ \quad k \text{ or } \bar{k} \text{ occurs in} \\ \quad B_1\, l^\bullet\, B_2 \;\|\; F, C \end{cases}$$

From Backtracking to Backjumping

Backjump

$$B_1 \, l^\bullet \, B_2 \parallel F, C \quad \to \quad B_1 \, k \parallel F, C \quad \textbf{if} \begin{cases} 1. \; B_1 \, l^\bullet \, B_2 \models \neg C, \\ 2. \; \text{for some clause } D \vee k \\ \quad F, C \models D \vee k, \\ \quad B_1 \models \neg D, \\ \quad k \text{ is undefined in } B_1, \\ \quad k \text{ or } \overline{k} \text{ occurs in} \\ \quad B_1 \, l^\bullet \, B_2 \parallel F, C \end{cases}$$

Whenever 1. holds, a *backjump clause* $D \vee k$ is computable from C

Basic DPLL System

At the core, current DPLL-based SAT solvers are implementations of the transition system:

Basic DPLL

- **Propagate**
- **Decide**
- **Fail**
- **Backjump**

Enhancements to Basic DPLL

Enhancements to Basic DPLL

Learn

$$B \parallel F \;\;\longrightarrow\;\; B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } F, \\ F \models C \end{cases}$$

Enhancements to Basic DPLL

Learn

$$B \parallel F \;\; \longrightarrow \;\; B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } F, \\ F \models C \end{cases}$$

Usually, C is a clause identified during conflict analysis

Enhancements to Basic DPLL

Learn

$$B \parallel F \quad \longrightarrow \quad B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } F, \\ F \models C \end{cases}$$

Forget

$$B \parallel F, C \quad \longrightarrow \quad B \parallel F \quad \textbf{if } F \models C$$

Enhancements to Basic DPLL

Learn

$$B \parallel F \;\longrightarrow\; B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } F, \\ F \models C \end{cases}$$

Forget

$$B \parallel F, C \;\longrightarrow\; B \parallel F \quad \textbf{if } F \models C$$

Restart

$$B \parallel F \;\longrightarrow\; \top \parallel F$$

Enhancements to Basic DPLL

Learn

$$B \parallel F \quad \longrightarrow \quad B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } F, \\ F \models C \end{cases}$$

Forget

$$B \parallel F, C \quad \longrightarrow \quad B \parallel F \quad \textbf{if } F \models C$$

Restart

$$B \parallel F \quad \longrightarrow \quad \top \parallel F$$

Modern DPLL = Basic DPLL + $\{$ **Learn**, **Forget**, **Restart** $\}$

Correctness of Abstract DPLL

Proposition For a rule application strategy to be fair it suffices to

- apply **Learn/Forget** only finitely many times,
- apply **Restart** only with increased periodicity, and
- stop with a state $B \parallel F$ only if
  - $B \models F$ or
  - F is irreducible by **Propagate**, **Decide** and **Backjump**

Correctness of Abstract DPLL

Proposition For a rule application strategy to be fair it suffices to

- apply **Learn/Forget** only finitely many times,
- apply **Restart** only with increased periodicity, and
- stop with a state $B \parallel F$ only if
  - $B \models F$ or
  - F is irreducible by **Propagate**, **Decide** and **Backjump**

This rather weak sufficient condition can be weakened further

Correctness of Abstract DPLL

Proposition For a rule application strategy to be fair it suffices to

- apply **Learn/Forget** only finitely many times,
- apply **Restart** only with increased periodicity, and
- stop with a state $B \parallel F$ only if
  - $B \models F$ or
  - F is irreducible by **Propagate**, **Decide** and **Backjump**

This rather weak sufficient condition can be weakened further

Proposition (recall) Fair strategies are terminating, sound, complete, proof confluent, and model finding

Clause Tableaux Modulo Theories

Let $\mathcal{T}$ be a theory with a decidable ground satisfiability problem

Clause Tableaux Modulo Theories

Let $\mathcal{T}$ be a theory with a decidable ground satisfiability problem

Assume we already have a $\mathcal{T}$*-solver*, a decision procedure for the $\mathcal{T}$-satisfiability of conjunctions of ground literals

Clause Tableaux Modulo Theories

Let $\mathcal{T}$ be a theory with a decidable ground satisfiability problem

Assume we already have a $\mathcal{T}$*-solver*, a decision procedure for the $\mathcal{T}$-satisfiability of conjunctions of ground literals

Then we can easily extend clause tableaux to deal with the full ground fragment

Clause Tableaux Modulo Theories

Let $\mathcal{T}$ be a theory with a decidable ground satisfiability problem

Assume we already have a $\mathcal{T}$*-solver*, a decision procedure for the $\mathcal{T}$-satisfiability of conjunctions of ground literals

Then we can easily extend clause tableaux to deal with the full ground fragment

We can do the same with DPLL, and capitalize on efficient DPLL engines

Clause Tableaux Modulo Theories

Let $\mathcal{T}$ be a theory with a decidable ground satisfiability problem

$\mathcal{T}$-Close

$T; B \parallel F \quad \longrightarrow \quad T \parallel F \quad \textbf{if } B$ is $\mathcal{T}$-inconsistent

B is *$\mathcal{T}$-(in)consistent* if the set of its literals is $\mathcal{T}$-(un)satisfiable

Clause Tableaux Modulo Theories

Let $\mathcal{T}$ be a theory with a decidable ground satisfiability problem

$\mathcal{T}$-Close

$$T;\, B \parallel F \quad \longrightarrow \quad T \parallel F \quad \textbf{if } B \text{ is } \mathcal{T}\text{-inconsistent}$$

Expand

$$T;\, B \parallel F, l_1 \vee \cdots \vee l_n \quad \longrightarrow \quad T;\, B\, l_1; \ldots; B\, l_n \parallel F, l_1 \vee \cdots \vee l_n \quad \textbf{if } (*)$$

$$(*) = \begin{cases} B \text{ is consistent (propositionally)} \\ B \not\models l_1 \vee \cdots \vee l_n \text{ (propositionally)} \end{cases}$$

Clause Tableaux Modulo Theories

Let $\mathcal{T}$ be a theory with a decidable ground satisfiability problem

$\mathcal{T}$-Close

$$T;\ B \parallel F \quad \rightarrow \quad T \parallel F \quad \textbf{if } B \text{ is } \mathcal{T}\text{-inconsistent}$$

Expand

$$T;\ B \parallel F, l_1 \vee \cdots \vee l_n \quad \rightarrow \quad T;\ B\,l_1;\ \ldots;\ B\,l_n \parallel F, l_1 \vee \cdots \vee l_n \quad \textbf{if } (*)$$

$$(*) = \begin{cases} B \text{ is consistent (propositionally)} \\ B \not\models l_1 \vee \cdots \vee l_n \text{ (propositionally)} \end{cases}$$

Empty

$$\emptyset \parallel F \quad \rightarrow \quad \mathit{fail}$$

Derivations Modulo $\mathcal{T}$

- A *derivation (of a clause set F)* is a $\rightarrow$-chain starting with $\top \parallel F$.
- A finite derivation $\top \parallel F \rightarrow \cdots \rightarrow S$ is *exhausted* if S
  - is $T; B \parallel G$ where B is $\mathcal{T}$-consistent and (propositionally) entails G, or
  - is irreducible by the rules.
- A rule application strategy is *fair* if it stops only with an exhausted derivation.

Proof Procedures as Rule Application Strategies

Proposition Every fair rule application strategy for ground clause tableaux modulo $\mathcal{T}$ is

- Terminating: it generates only finite derivations.
- Sound: it generates a derivation $\top \parallel F \longrightarrow \cdots \longrightarrow \mathit{fail}$ only if F is $\mathcal{T}$-unsatisfiable.
- Complete: it can generate a derivation $\top \parallel F \longrightarrow \cdots \longrightarrow \mathit{fail}$ if F is $\mathcal{T}$-unsatisfiable.
- Proof confluent: it can extend any derivation of $\top \parallel F$ with a $\mathcal{T}$-unsatisfiable F to one ending in fail.
- Model finding: it stops with state $T \parallel G$ only if a branch of T is a $\mathcal{T}$-consistent (propositional) model of F.

Abstract DPLL Modulo Theories

Works with any DPLL engine and $\mathcal{T}$-solver but is best with

1. an on-line DPLL engine and
2. an incremental $\mathcal{T}$-solver

Abstract DPLL Modulo Theories

Works with any DPLL engine and $\mathcal{T}$-solver but is best with

1. an on-line DPLL engine and
2. an incremental $\mathcal{T}$-solver

It consists of the following rules:

- **Propagate**, **Decide**, **Fail**, **Restart**
  (as in the propositional case) and
- $\mathcal{T}$-**Backjump**, $\mathcal{T}$-**Learn**, $\mathcal{T}$-**Forget**
  (theory versions of **Backjump**, **Learn**, **Forget**, resp.)

Theory Rules

$\mathcal{T}$-Backjump

$$B_1\, l^\bullet\, B_2 \;\|\; F, C \quad \longrightarrow \quad B_1\, k \;\|\; F, C \quad \textbf{if} \begin{cases} 1.\ B_1\, l^\bullet\, B_2 \models \neg C, \\ 2.\ \text{for some clause } D \vee k \\ \quad F, C \models_{\mathcal{T}} D \vee k, \\ \quad B_1 \models \neg D, \\ \quad k \text{ is undefined in } M, \\ \quad k \text{ or } \overline{k} \text{ occurs in} \\ \quad B_1\, l^\bullet\, B_2 \;\|\; F, C \end{cases}$$

Not.: $F \models_{\mathcal{T}} G$ iff every model of $\mathcal{T}$ that satisfies F satisfies G

Theory Rules

$\mathcal{T}$-Backjump

$$B_1\, l^\bullet\, B_2 \parallel F, C \quad \longrightarrow \quad B_1\, k \parallel F, C \quad \textbf{if} \begin{cases} 1.\ B_1\, l^\bullet\, B_2 \models \neg C, \\ 2.\ \text{for some clause } D \vee k \\ \quad F, C \models_{\mathcal{T}} D \vee k, \\ \quad B_1 \models \neg D, \\ \quad k \text{ is undefined in } M, \\ \quad k \text{ or } \overline{k} \text{ occurs in} \\ \quad B_1\, l^\bullet\, B_2 \parallel F, C \end{cases}$$

$\mathcal{T}$-Learn

$$B \parallel F \quad \longrightarrow \quad B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } B \parallel F, \\ F \models_{\mathcal{T}} C \end{cases}$$

$\mathcal{T}$-Forget

$$B \parallel F, C \quad \longrightarrow \quad B \parallel F \quad \textbf{if } F \models_{\mathcal{T}} C$$

Correctness of Abstract DPLL Modulo Theories

Proposition For a rule application strategy to be fair it suffices to

- apply $\mathcal{T}$**-Learn**/$\mathcal{T}$**-Forget** only finitely many times,
- apply **Restart** only with increased periodicity, and
- stop with a state $B \parallel F$ only if B is $\mathcal{T}$-consistent and
  - $B \models F$ or
  - F is irreducible by **Propagate**, **Decide** and $\mathcal{T}$**-Backjump**

From Complete to Incomplete Theory Solvers

Recall: On reaching a state $B \parallel G$ with $B \models G$, the $\mathcal{T}$-solver must determine whether $B \models_{\mathcal{T}} \bot$

From Complete to Incomplete Theory Solvers

Recall: On reaching a state $B \parallel G$ with $B \models G$, the $\mathcal{T}$-solver must determine whether $B \models_{\mathcal{T}} \bot$

- At the very least, the $\mathcal{T}$-solver must be refutationally sound:

  never calling a $\mathcal{T}$-satisfiable set B of literals $\mathcal{T}$-unsatisfiable,

From Complete to Incomplete Theory Solvers

Recall: On reaching a state $B \parallel G$ with $B \models G$, the $\mathcal{T}$-solver must determine whether $B \models_{\mathcal{T}} \bot$

- At the very least, the $\mathcal{T}$-solver must be refutationally sound:

  never calling a $\mathcal{T}$-satisfiable set B of literals $\mathcal{T}$-unsatisfiable,

- Ideally, it should also be refutationally complete:

From Complete to Incomplete Theory Solvers

Recall: On reaching a state $B \parallel G$ with $B \models G$, the $\mathcal{T}$-solver must determine whether $B \models_{\mathcal{T}} \bot$

- At the very least, the $\mathcal{T}$-solver must be refutationally sound:

  never calling a $\mathcal{T}$-satisfiable set B of literals $\mathcal{T}$-unsatisfiable,

- Ideally, it should also be refutationally complete:

  always able to recognize a $\mathcal{T}$-unsatisfiable set B of literals as such.

From Complete to Incomplete Theory Solvers

Recall: On reaching a state $B \parallel G$ with $B \models G$, the $\mathcal{T}$-solver must determine whether $B \models_{\mathcal{T}} \bot$

- At the very least, the $\mathcal{T}$-solver must be refutationally sound:

  never calling a $\mathcal{T}$-satisfiable set B of literals $\mathcal{T}$-unsatisfiable,

- Ideally, it should also be refutationally complete:

  always able to recognize a $\mathcal{T}$-unsatisfiable set B of literals as such.

- For certain theories, it is advantageous to relax the refutational completeness requirement.

Case Splitting

- For certain theories, determining that B is $\mathcal{T}$-unsatisfiable requires reasoning by cases.

Case Splitting

- For certain theories, determining that B is $\mathcal{T}$-unsatisfiable requires reasoning by cases.

Example: $\mathcal{T}$ = the theory of arrays.

$$B = \{\underbrace{r(w(a,i,x),j) \neq x}_{1},\ \underbrace{r(w(a,i,x),j) \neq r(a,j)}_{2}\}$$

Case Splitting

- For certain theories, determining that B is $\mathcal{T}$-unsatisfiable requires reasoning by cases.

Example: $\mathcal{T}$ = the theory of arrays.

$$B = \{\underbrace{r(w(a,i,x),j) \neq x}_{1},\ \underbrace{r(w(a,i,x),j) \neq r(a,j)}_{2}\}$$

$i = j$) Then, $r(w(a,i,x),j) = x$. Contradiction with 1.

Case Splitting

- For certain theories, determining that B is $\mathcal{T}$-unsatisfiable requires reasoning by cases.

Example: $\mathcal{T}$ = the theory of arrays.

$$B = \{\underbrace{r(w(a,i,x),j) \neq x}_{1},\ \underbrace{r(w(a,i,x),j) \neq r(a,j)}_{2}\}$$

$i = j$) Then, $r(w(a,i,x),j) = x$. Contradiction with 1.

$i \neq j$) Then, $r(w(a,i,x),j) = r(a,j)$. Contradiction with 2.

Case Splitting

- For certain theories, determining that B is $\mathcal{T}$-unsatisfiable requires reasoning by cases.

Example: $\mathcal{T}$ = the theory of arrays.

$$B = \{\underbrace{r(w(a,i,x),j) \neq x}_{1},\ \underbrace{r(w(a,i,x),j) \neq r(a,j)}_{2}\}$$

$i = j$) Then, $r(w(a,i,x),j) = x$. Contradiction with 1.

$i \neq j$) Then, $r(w(a,i,x),j) = r(a,j)$. Contradiction with 2.

Conclusion: B is $\mathcal{T}$-unsatisfiable.

Case Splitting

- For certain theories, determining that B is $\mathcal{T}$-unsatisfiable requires reasoning by cases

Case Splitting

- For certain theories, determining that B is $\mathcal{T}$-unsatisfiable requires reasoning by cases
- A complete $\mathcal{T}$-solver does that with internal case splitting and backtracking mechanisms
(essentially implementing a ground tableaux calculus with theory specific expansion rules)

Case Splitting

- For certain theories, determining that B is $\mathcal{T}$-unsatisfiable requires reasoning by cases
- A complete $\mathcal{T}$-solver does that with internal case splitting and backtracking mechanisms
  (essentially implementing a ground tableaux calculus with theory specific expansion rules)
- A more economical approach is to lift case splitting from the $\mathcal{T}$-solver to the DPLL engine

Case Splitting

- For certain theories, determining that B is $\mathcal{T}$-unsatisfiable requires reasoning by cases
- A complete $\mathcal{T}$-solver does that with internal case splitting and backtracking mechanisms
(essentially implementing a ground tableaux calculus with theory specific expansion rules)
- A more economical approach is to lift case splitting from the $\mathcal{T}$-solver to the DPLL engine
- Basic idea: Code each case split as a set of clauses and send them as needed to the engine so it can split on them

Splitting on Demand [?]

Basic idea: Code each case split as a set of clauses and send them as needed to the engine so it can split on them.

Possible benefits:

- All case-splitting is coordinated by the DPLL engine
- Only have to implement case-splitting infrastructure in one place
- DPLL heuristics are not sabotaged by internal theory splitting

Splitting on Demand [?]

Basic idea: Code each case split as a set of clauses and send them as needed to the engine so it can split on them.

Splitting on Demand [?]

Basic idea: Code each case split as a set of clauses and send them as needed to the engine so it can split on them.

Basic Scenario:

$$B = \{\ldots,\ s = \underbrace{r(w(a,i,t),j)}_{s'},\ \ldots,\}$$

Splitting on Demand [?]

Basic idea: Code each case split as a set of clauses and send them as needed to the engine so it can split on them.

Basic Scenario:

$$B = \{\ldots,\ s = \underbrace{r(w(a,i,t),j)}_{s'},\ \ldots,\}$$

DPLL Engine: “Is B $\mathcal{T}$-unsatisfiable?”

Splitting on Demand [?]

Basic idea: Code each case split as a set of clauses and send them as needed to the engine so it can split on them.

Basic Scenario:

$$B = \{\ldots,\ s = \underbrace{r(w(a, i, t), j)}_{s'},\ \ldots,\}$$

DPLL Engine: "Is B $\mathcal{T}$-unsatisfiable?"

$\mathcal{T}$**-solver:** "I do not know yet, but it will help me if you split on these theory lemmas:

$$s = s' \wedge i = j \rightarrow s = t, \quad s = s' \wedge i \neq j \rightarrow s = r(a, j)$$ "

Splitting on Demand in Abstract DPLL

How do we extend ADPLL Modulo Theories to handle such theory case-splits?

Splitting on Demand in Abstract DPLL

How do we extend ADPLL Modulo Theories to handle such theory case-splits?

Recall the $\mathcal{T}$**-Learn** rule:

$$B \parallel F \implies B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } B \parallel \ldots \\ F \models_{\mathcal{T}} C \end{cases}$$

Splitting on Demand in Abstract DPLL

How do we extend ADPLL Modulo Theories to handle such theory case-splits?

Recall the $\mathcal{T}$**-Learn** rule:

$$B \parallel F \quad \Longrightarrow \quad B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } B \parallel F \\ F \models_{\mathcal{T}} C \end{cases}$$

This rule allows a theory solver to send clauses to the DPLL engine as long as their atoms occur in $B \parallel F$.

Splitting on Demand in Abstract DPLL

How do we extend ADPLL Modulo Theories to handle such theory case-splits?

Recall the $\mathcal{T}$-**Learn** rule:

$$B \parallel F \quad \Longrightarrow \quad B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } B \parallel F \\ F \models_{\mathcal{T}} C \end{cases}$$

This rule allows a theory solver to send clauses to the DPLL engine as long as their atoms occur in $B \parallel F$.

We wish to relax this requirement to allow additional atoms, possibly even containing new terms.

Splitting on Demand in Abstract DPLL

It is enough to replace $\mathcal{T}$-**Learn** with

Extended $\mathcal{T}$-**Learn**

$$B \parallel F \quad \rightarrow \quad B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur} \\ \text{in } F \text{ or in } \mathcal{L}(B), \\ F \models_{\mathcal{T}} \gamma_F(C) \end{cases}$$

Splitting on Demand in Abstract DPLL

It is enough to replace $\mathcal{T}$-**Learn** with

Extended $\mathcal{T}$-**Learn**

$$B \parallel F \quad \rightarrow \quad B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur} \\ \text{in } F \text{ or in } \mathcal{L}(B), \\ F \models_{\mathcal{T}} \gamma_F(C) \end{cases}$$

where:

$\gamma_F(C)$ existentially quantifies the free constants of C not occurring in F.

Splitting on Demand in Abstract DPLL

It is enough to replace $\mathcal{T}$-**Learn** with

Extended $\mathcal{T}$-**Learn**

$$B \parallel F \;\; \rightarrow \;\; B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur} \\ \text{in } F \text{ or in } \mathcal{L}(B), \\ F \models_{\mathcal{T}} \gamma_F(C) \end{cases}$$

where:

$\mathcal{L}$ is a mapping from literal sets to literal sets such that

1. $B \subseteq \mathcal{L}(B)$.
2. If $B \subseteq B'$, then $\mathcal{L}(B) \subseteq \mathcal{L}(B')$.
3. $\mathcal{L}(\mathcal{L}(B)) = \mathcal{L}(B)$.

Splitting on Demand in Abstract DPLL

It is enough to replace $\mathcal{T}$-**Learn** with

Extended $\mathcal{T}$-**Learn**

$$B \parallel F \;\; \longrightarrow \;\; B \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur} \\ \text{in } F \text{ or in } \mathcal{L}(B), \\ F \models_{\mathcal{T}} \gamma_F(C) \end{cases}$$

Fact: For many theories with a theory solver, such an $\mathcal{L}$ exists.

Note: The set $\mathcal{L}(B)$ never needs to be computed explicitly.

Extending Abstract DPLL Modulo Theories

Now we can relax the requirement on the theory solver:

In the state $B \parallel G$, if $B \models G$, the theory solver must either

- determine whether $B \models_{\mathcal{T}} \bot$ or
- generate a new clause by $\mathcal{T}$**-Learn** containing at least one literal of $\mathcal{L}(B)$ undefined in B.

Extending Abstract DPLL Modulo Theories

Now we can relax the requirement on the theory solver:

In the state $B \parallel G$, if $B \models G$, the theory solver must either

- determine whether $B \models_{\mathcal{T}} \bot$ or
- generate a new clause by $\mathcal{T}$**-Learn** containing at least one literal of $\mathcal{L}(B)$ undefined in B.

Note: the $\mathcal{T}$-solver is required to determine $B \models_{\mathcal{T}} \bot$ only if all literals in $\mathcal{L}(B)$ are defined in B.

Extending Abstract DPLL Modulo Theories

Now we can relax the requirement on the theory solver:

In the state $B \parallel G$, if $B \models G$, the theory solver must either

- determine whether $B \models_{\mathcal{T}} \bot$ or
- generate a new clause by $\mathcal{T}$**-Learn** containing at least one literal of $\mathcal{L}(B)$ undefined in B.

Note: the $\mathcal{T}$-solver is required to determine $B \models_{\mathcal{T}} \bot$ only if all literals in $\mathcal{L}(B)$ are defined in B.

In practice, to determine if $B \models_{\mathcal{T}} \bot$ the $\mathcal{T}$-solver only needs a small subset of $\mathcal{L}(B)$ to be defined in B.

Correctness Results

Given the new rules, previous correctness results can be easily extended.

- Soundness: Holds because the new $\mathcal{T}$**-Learn** rule is $\mathcal{T}$-satisfiability preserving (even if not $\mathcal{T}$-equivalence preserving)

Correctness Results

Given the new rules, previous correctness results can be easily extended.

- Soundness: Holds because the new $\mathcal{T}$**-Learn** rule is $\mathcal{T}$-satisfiability preserving (even if not $\mathcal{T}$-equivalence preserving)
- Completeness: Holds as long as the theory solver decides $B \models_{\mathcal{T}} \perp$ whenever all literals in $\mathcal{L}(F)$ are defined

Correctness Results

Given the new rules, previous correctness results can be easily extended.

- Soundness: Holds because the new $\mathcal{T}$**-Learn** rule is $\mathcal{T}$-satisfiability preserving (even if not $\mathcal{T}$-equivalence preserving)
- Completeness: Holds as long as the theory solver decides $B \models_{\mathcal{T}} \bot$ whenever all literals in $\mathcal{L}(F)$ are defined
- Termination: Holds under the same conditions as the original system (because $\mathcal{L}(F)$ is finite)

Thank you