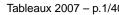


# An Abstract Framework for Satisfiability Modulo Theories

Cesare Tinelli

The University of Iowa





### **Based on joint work with:**

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

- **Special thanks to:** 
  - the TABLEAUX 2007 PC for the invitation.



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

- 6 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:

- 6 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:
  - A Hardware verification: theory of equality, of bit vectors.

- 6 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:
  - A Hardware verification: theory of equality, of bit vectors.
  - Timed automata, planning: theory of integers/reals.

- 6 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.

- 6 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.
- 6 We refer to this general problem as (ground) Satisfiability Modulo Theories, or SMT.

Satisfiability Modulo a Theory  $\mathcal{T}$ 

Ground T-satisfiability problem for a theory T:

Is there a model of  $\mathcal T$  that satisfies a given ground formula  $\varphi$  ?

Satisfiability Modulo a Theory  ${\mathcal T}$ 

Ground T-satisfiability problem for a theory T:

Is there a model of  $\mathcal T$  that satisfies a given ground formula  $\varphi$  ?

Some popular theories

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



6 Note: The T-satisfiability of ground formulas is decidable iff the T-satisfiability of sets of literals is decidable



- 6 Note: The T-satisfiability of ground formulas is decidable iff the 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



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



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





- 6 Eager approach [CBMC, UCLID, ...]:
  - **translate**  $\varphi$  into an equisat. propositional formula,
  - △ feed it to any SAT solver.

- *Eager approach* [CBMC, UCLID, ...]:
  - translate  $\varphi$  into an equisat. propositional formula,
  - feed it to any SAT solver.
- Lazy approach [Barcelogic, CVC\*, ICS, MathSAT, Verifun, Yices, Z3, ...]:
  - treat  $\varphi$  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.

- 6 Eager approach [CBMC, UCLID, ...]:
  - translate  $\varphi$  into an equisat. propositional formula,
  - feed it to any SAT solver.
- Lazy approach [Barcelogic, CVC\*, ICS, MathSAT, Verifun, Yices, Z3, ...]:
  - treat  $\varphi$  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.
- 6 This talk focuses on the lazy approach.



#### Lazy approach:

- 6 treat  $\varphi$  as a propositional formula,
- 6 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.



Skip over implementation details and unimportant control aspects.



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



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



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



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



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



#### Grand claim of the day:

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

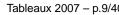




#### Grand claim of the day:

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



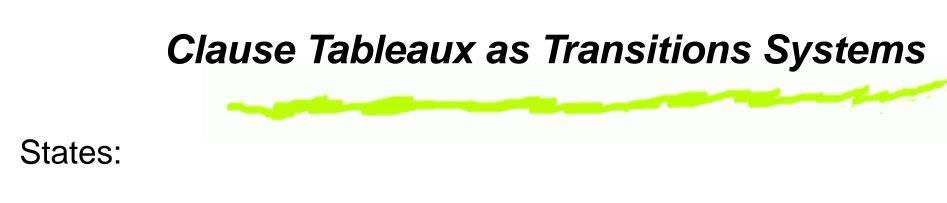
### Clause Tableaux as Transitions Systems

States:

fail or  $T \parallel F$ 

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.





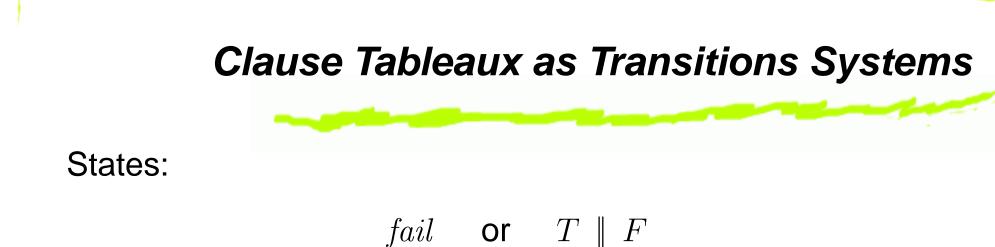
fail or  $T \parallel F$ 

#### Initial state:

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

#### **Expected final states:**

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



#### Notation:

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

#### Convention:

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

### Transition Rules for a Basic Clause Tableau

### Close $T; B \parallel F \rightarrow T \parallel F \text{ if } B \text{ is inconsistent (i.e., } p, \neg p \in B)$

### Transition Rules for a Basic Clause Tableau

### Close $T; B \parallel F \rightarrow T \parallel F \text{ if } B \text{ is inconsistent (i.e., } p, \neg p \in B)$

### Expand

$$T; B \parallel F, l_1 \lor \cdots \lor l_n \to T; B l_1; \ldots; B l_n \parallel F, l_1 \lor \cdots \lor l_n \quad \text{if } (*)$$
$$(*) = \begin{cases} B \text{ is consistent} \\ B \not\models l_1 \lor \cdots \lor l_n \end{cases}$$

### Transition Rules for a Basic Clause Tableau

### Close $T; B \parallel F \rightarrow T \parallel F \text{ if } B \text{ is inconsistent (i.e., } p, \neg p \in B)$

### Expand

$$T; B \parallel F, l_1 \lor \cdots \lor l_n \to T; B l_1; \ldots; B l_n \parallel F, l_1 \lor \cdots \lor l_n \quad \text{if } (*)$$
$$(*) = \begin{cases} B \text{ is consistent} \\ B \not\models l_1 \lor \cdots \lor l_n \end{cases}$$

 $\begin{array}{cccc} \mathbf{Empty} \\ \emptyset \parallel F & \to & fail \end{array}$ 

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

### **Proof Procedures as Rule Application Strategies**

- 6 A *derivation (of a clause set* F) is a  $\rightarrow$ -chain starting with  $\top \parallel F$ .
- 6 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.
- 6 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:

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

### **Enhancements to Basic Clause Tableaux**

#### **Additional rules**

### Conflict

### $T; B \parallel F, C \rightarrow T \parallel F, C \quad \text{if } B \models \neg C$

C is a *conflicting* clause

# **Enhancements to Basic Clause Tableaux**

## **Additional rules**

## Conflict

 $T; B \parallel F, C \rightarrow T \parallel F, C \quad \text{if } B \models \neg C$ 

C is a *conflicting* clause

#### Propagate

 $T; B \parallel F, C \lor l \rightarrow T; B l \parallel F, C \lor l \text{ if } \begin{cases} B \models \neg C \\ l \text{ is undefined in } B \end{cases}$ 



# **Enhancements to Basic Clause Tableaux**

### **Additional rules**

#### Conflict

 $T; B \parallel F, C \rightarrow T \parallel F, C \quad \text{if } B \models \neg C$ 

C is a *conflicting* clause

#### Propagate

 $T; B \parallel F, C \lor l \rightarrow T; B l \parallel F, C \lor l \text{ if } \begin{cases} B \models \neg C \\ l \text{ is undefined in } B \end{cases}$ 

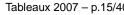
#### Split (atomic cut)

 $T; B \parallel F \rightarrow T; Bl; B\overline{l} \parallel F \text{ if } \begin{cases} l \text{ or } \overline{l} \text{ occurs in } F, \\ l \text{ is undefined in } B \end{cases}$ 





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





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



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



#### **Split**

# $T; B \parallel F \rightarrow T; Bl; B\overline{l} \parallel F \text{ if } \begin{cases} l \text{ or } l \text{ occurs in } F, \\ l \text{ is undefined in } B \end{cases}$

#### becomes

#### Decide

$$B \parallel F \rightarrow Bl^{\bullet} \parallel F \text{ if } \begin{cases} l \text{ or } \overline{l} \text{ occurs in } F, \\ l \text{ is undefined in } B \end{cases}$$

Notation: *l*• is *l* annotated as a *decision literal* 

# **Enhancements to Basic Clause Tableaux**

# **Conflict** $T; B \parallel F, C \rightarrow T \parallel F, C \text{ if } B \models \neg C$

#### becomes

## Backtrack

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

# **Enhancements to Basic Clause Tableaux**



 $\begin{array}{cccc} \mathbf{Empty} \\ \emptyset \parallel F & \to & fail \end{array}$ 

#### becomes

#### Fail

# $B \parallel F, C \rightarrow fail \text{ if } \begin{cases} B \models \neg C, \\ B \text{ contains no decision literals} \end{cases}$

# **Our Abstract Version of the Original DPLL**

#### Propagate

$$B \parallel F, C \lor l \rightarrow B, l \parallel F, C \lor l \quad \text{if } \begin{cases} B \models \neg C \\ l \text{ is undefined in } B \end{cases}$$

**Decide** 
$$B \parallel F \rightarrow Bl^{\bullet} \parallel F$$
 if  $\begin{cases} l \text{ or } \overline{l} \text{ occurs in } F, \\ l \text{ is undefined in } B \end{cases}$ 

#### Fail

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

#### **Backtrack**

 $B_1 l^{\bullet} B_2 \parallel F, C \rightarrow B_1 \overline{l} \parallel F, C \text{ if } \begin{cases} B_1 l^{\bullet} B_2 \models \neg C, \\ l \text{ last decision literal} \end{cases}$ 

# Smarter Backtracking

#### **Backtrack**

$$B_1 l^{\bullet} B_2 \parallel F, C \rightarrow B_1 \overline{l} \parallel F, C \text{ if } \begin{cases} B_1 l^{\bullet} B_2 \models \neg C, \\ l \text{ last decision literal} \end{cases}$$

# is replaced in modern implementations by

# Backjump

$$B_1 l^{\bullet} B_2 \parallel F, C \rightarrow B_1 k \parallel F, C \text{ if }$$

1. 
$$B_1 l^{\bullet} B_2 \models \neg C$$
,  
2. for some clause  $D \lor k$   
 $F, C \models D \lor k$ ,  
 $B_1 \models \neg D$ ,  
 $k$  is undefined in  $B_1$ ,  
 $k$  or  $\overline{k}$  occurs in  
 $B_1 l^{\bullet} B_2 \parallel F, C$ 



# Backjump

## $B_1 l^{\bullet} B_2 \parallel F, C \rightarrow B_1 k \parallel F, C \text{ if } \langle \rangle$

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

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



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

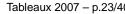
# **Basic DPLL**

- 6 Propagate
- 6 Decide
- 6 Fail
- 6 Backjump





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





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

Usually, C is a clause identified during conflict analysis





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

# Forget $B \parallel F, C \rightarrow B \parallel F \text{ if } F \models C$



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

# Forget $B \parallel F, C \rightarrow B \parallel F \text{ if } F \models C$

#### Restart

 $B \parallel F \rightarrow \top \parallel F$ 



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

# Forget $B \parallel F, C \rightarrow B \parallel F \text{ if } F \models C$

## Restart

 $B \parallel F \rightarrow \top \parallel F$ 

Modern DPLL = Basic DPLL + { Learn, Forget, Restart }

# **Correctness of Abstract DPLL**

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

- 6 apply Learn/Forget only finitely many times,
- 6 apply **Restart** only with increased periodicity, and
- **6** 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

- 6 apply Learn/Forget only finitely many times,
- 6 apply **Restart** only with increased periodicity, and
- **5** 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

- 6 apply Learn/Forget only finitely many times,
- 6 apply **Restart** only with increased periodicity, and
- **5** 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

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

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

Assume we already have a T-solver, a decision procedure for the T-satisfiability of conjunctions of ground literals

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

Assume we already have a T-solver, a decision procedure for the T-satisfiability of conjunctions of ground literals

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

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

Assume we already have a T-solver, a decision procedure for the 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

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

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

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

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

# $\mathcal{T}\text{-}Close$

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

## Expand

 $T; B \parallel F, l_1 \lor \cdots \lor l_n \rightarrow T; B l_1; \ldots; B l_n \parallel F, l_1 \lor \cdots \lor l_n \quad \text{if } (*)$  $(*) = \begin{cases} B \text{ is consistent (propositionally)} \\ B \not\models l_1 \lor \cdots \lor l_n \text{ (propositionally)} \end{cases}$ 

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

# $\mathcal{T}\text{-}Close$

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

## Expand

 $T; B \parallel F, l_1 \lor \cdots \lor l_n \rightarrow T; B l_1; \ldots; B l_n \parallel F, l_1 \lor \cdots \lor l_n \quad \text{if } (*)$  $(*) = \begin{cases} B \text{ is consistent (propositionally)} \\ B \not\models l_1 \lor \cdots \lor l_n \text{ (propositionally)} \end{cases}$ 

#### Empty

 $\emptyset \parallel F \longrightarrow fail$ 



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

# **Proof Procedures as Rule Application Strategies**

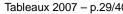
**Proposition** Every fair rule application strategy for ground clause tableaux modulo T is

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



#### Works with any DPLL engine and T-solver but is best with

- 1. an on-line DPLL engine and
- 2. an incremental T-solver





Works with any DPLL engine and T-solver but is best with

- 1. an on-line DPLL engine and
- 2. an incremental T-solver

It consists of the following rules:

- Propagate, Decide, Fail, Restart
   (as in the propositional case) and
- 5 T-Backjump, T-Learn, T-Forget (theory versions of Backjump, Learn, Forget, resp.)

# **Theory Rules**

#### $\mathcal{T}$ -Backjump

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

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

# Theory Rules

#### T-Backjump

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

#### $\mathcal{T}$ -Learn

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

# T-Forget

 $B \parallel F, C \rightarrow B \parallel F \text{ if } F \models_{\mathcal{T}} C$ 

# **Correctness of Abstract DPLL Modulo Theories**

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

- 6 apply T-Learn/T-Forget only finitely many times,
- 6 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
     T-Backjump

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

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

6 At the very least, the T-solver must be refutationally sound:

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

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

6 At the very least, the T-solver must be refutationally sound:

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

6 Ideally, it should also be refutationally complete:

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

6 At the very least, the 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 *T*-unsatisfiable set *B* of literals as such.

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

6 At the very least, the 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 *T*-unsatisfiable set *B* of literals as such.
- 6 For certain theories, it is advantageous to relax the refutational completeness requirement.





**Example:** T = the theory of arrays.

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



**Example:** 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.



**Example:** 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.



**Example:** 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 T-unsatisfiable.





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



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



- 6 For certain theories, determining that B is T-unsatisfiable requires reasoning by cases
- A complete T-solver does that with internal case splitting and backtracking mechanisms (essentially implementing a ground tableaux calculus with theory specific expansion rules)
- 6 A more economical approach is to lift case splitting from the T-solver to the DPLL engine
- 6 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:

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





Basic Scenario:

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



Basic Scenario:

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

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



Basic Scenario:

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

**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:

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

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

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

Recall the T-Learn rule:

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

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

Recall the T-Learn rule:

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

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

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

Recall the T-Learn rule:

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

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.

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

Extended T-Learn

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

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

```
Extended T-Learn
```

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

where:

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

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

```
Extended T-Learn
```

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

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.  $\mathcal{L}(\mathcal{L}(B)) = \mathcal{L}(B)$ .

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

```
Extended T-Learn
```

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

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

- 6 determine whether  $B \models_{\mathcal{T}} \bot$  or
- 6 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

- 6 determine whether  $B \models_{\mathcal{T}} \bot$  or
- 6 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

- 6 determine whether  $B \models_{\mathcal{T}} \bot$  or
- 6 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)
- 6 Completeness: Holds as long as the theory solver decides  $B \models_{\mathcal{T}} \bot$  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)
- 6 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

