Automated Theorem Proving: Satisfiability Modulo Theories

**PartiallyClips**

PLAYING "MINESWEEPER" AGAIN?

JUST WHILE THEY UPGRADE THE SERVER.

DUNNO WHAT YOU SEE IN THAT GAME.

IT'S FANTASTIC. THEY SHOULD TEACH IT FOR CREDIT IN SCHOOLS.

YOU'RE KIDDING, RIGHT?

DEAD SERIOUS. IT TEACHES LOGICAL REASONING.

HOW TO EXTRAPOLATE A CONCLUSION FROM KNOWN PREMISES. WHAT CONSTITUTES PROOF. HOW NOT TO DELUDE YOURSELF. THE WORLD WOULD BE A LOT BETTER OFF IF WE TaUGHT THIS STUFF TO OUR KIDS.

WELL, I CAN THINK OF SEVERAL COUNTRIES WHERE SCHOOL KIDS LEARN TO SWEEP MINES, AND THEY DON'T SEEM TOO WELL-OFF TO ME.

Yeah, ha-ha very poignant. But I'm saying if the whole world learned to think logically, there wouldn't be any mines to sweep.
One-Slide Summary

- An **automated theorem prover** is an algorithm that determines whether a mathematical or logical proposition is **valid** (satisfiable).

- A **theory** is a set of sentences with a deductive system that can determine satisfiability.

- A **satisfiability modulo theories** (SMT) instance is a proposition that can include logical connectives, equality, and terms from various theories.

- The theory of **equality** can be decided via **congruence closure** using **union-find**.

- **DPLL(T)** is an SMT algorithm that uses a modified DPLL SAT solver, a well-defined interface for Theories, and a **mapping** between propositional variables and Theory literals.

- We can use logical rules of inference to encode **proofs**. Proof checking is like type checking.
Combined Motivation

• We have seen:
  - How to handle \((A \ || \ !B) \Rightarrow (!A \ || \ C)\)
    • Satisfied by \{A, C\}, for example
    • Arbitrary boolean expressions of boolean variables
  - How to handle \((x + y \leq 5) \&\& (2x \geq 10)\)
    • Satisfied by \{x = 5, y = 0\}, for example
    • Conjunctions of linear inequalities of real variables

• But what about:
  - \((\text{strlen}(x) + y \leq 5) \Rightarrow (\text{strcat}(x,x) \neq \text{“abba”})\)
    • Satisfied by \{x = \text{“abc”}, y = 3\}, for example
High-Level Approach

• Beyond basic logic, we want to reason about
  - **Strings**: strlen(x), regexp_match(x, “[0-9]+”), …
  - **Equality**: a = b => f(a) = f(b), …
  - **Linear Arithmetic**: 2x + 3y <= 10, …
  - **Bitvectors**: (x >> 2) | y == 0xff, …
  - **Lists**: head(cons(p,q)) = p

• **All at the same time!**

• We will handle each domain separately (as a theory) and then combine them all together using DPLL and SAT as the “glue”.

Overall Plan

• Theory Introduction
• Theory of Equality
  - Congruence Closure
• The DPLL(T) solver for SMT
  - Formal Theory Interface
  - Changes to DPLL
• Proof and Proof Checking
I Have a Theory: It Could Be Bunnies

- In general, a **theory** is a set of sentences (**syntax**) with a deductive system that can determine satisfiability (**semantics**).

- Usually, the set of sentences is formally defined by a grammar of **terms** over **atoms**. The **satisfying assignment** (or **model**, or **interpretation**) maps **literals** (terms or negated terms) to booleans.

- We will consider theories that reason about **conjunctions** of literals.
Theory of Linear Inequalities

- Given a finite set of variables $V$ and a finite set of real-valued constants $C$

- Term ::= $C_1 V_1 + \ldots + C_n V_n \leq C_{n+1}$

- Conjunctions of terms can be decided via the Simplex method
Theory of Equality and Uninterpreted Functions

- Given a finite set of predicate symbols $P$ (each of which has an associated *arity*)
- Predicate ::= $P_i$ (Predicate$_1$, ... Predicate$_n$)
- Term ::= Predicate
  | Predicate$_1$ = Predicate$_2$

- Term Examples:
  - $f(g(a,b),a)$  // a, b have arity 0 (constants)
  - $f(a) = f(b)$  // f has arity 1
  - $f(f(x)) = f(x)$  // g has arity 2
Theory of Equality Definition

• Theory of \textit{equality with uninterpreted functions}

• Symbols: =, \neq, f, g, ...

• Axiomatically defined (A,B,C \not\equiv \text{Predicates}):

\[
\begin{array}{cccc}
\text{A=A} & \text{B=A} & \text{A=B} & \text{B=C} \\
\text{A=B} & \text{A=C} & \text{A=B} & \text{f(A) = f(B)}
\end{array}
\]

• Reflexive, Symmetric, Transitive, Definition of A Function (Extensionality)
Solving Equality

- Consider this conjunction of literals in the theory of equality:
  
  \[ g(g(g(x))) = x \land g(g(g(g(g(x))))) = x \land g(x) \neq x \]

- Is it satisfiable?
Solving Equality

• Consider this conjunction of literals in the theory of equality:

\[ g(g(g(x)))=x \land g(g(g(g(g(x))))))=x \land g(x) \neq x \]

• Is it satisfiable? No.

  4. \( gggg(x) = g(x) \) by extensionality (1)
  5. \( ggggg(x) = gg(x) \) by extensionality (4)
  6. \( x = gg(x) \) by transitivity (2, 5)
  7. \( g(x) = ggg(x) \) by extensionality (6)
  8. \( g(x) = x \) by transitivity (1, 7)
  9. false by contradiction (8, 3)
Tracking Equality

• To decide the theory of equality mechanically, we track equivalence classes of predicates

• Two operations
  - Set Equal (Union)
  - Check Equal (Find Parents)

• Basic approach: build a tree

  • A = B
  • D = E
  • C = B
Tracking Equality

• To decide the theory of equality mechanically, we track equivalence classes of predicates

• Two operations
  - Set Equal (Union)
  - Check Equal (Find Parents)

• Basic approach: build a tree

  • $A = B$
  • $D = E$
  • $C = B$
Tracking Equality

- To decide the theory of equality mechanically, we track equivalence classes of predicates
- Two operations
  - Set Equal (Union)
  - Check Equal (Find Parents)
- Basic approach: build a tree
  - $A = B$
  - $D = E$
  - $C = B$
Tracking Equality

• To decide the theory of equality mechanically, we track equivalence classes of predicates

• Two operations
  - Set Equal (Union)
  - Check Equal (Find Parents)

• Basic approach: build a tree
  
  A = B
  D = E
  C = B
Analogy: Raindrops Merging
Two Optimizations

let **union** x y =

let x_root, y_root = find x, find y in
if x_root = y_root then return ()
if x_root.depth < y_root.depth
  x_root.parent := y_root
else if y_root.depth < x_root.depth
  y_root.parent := x_root
else
  y_root.parent := x_root

x_root.depth := x_root.depth + 1

Keep trees short! Add smaller tree to bigger tree. Only increases total Depth if depths were equal!
Analogy: Forwarding vs. Change of Address
Two Optimizations

```ocaml
let rec find x =
  if x.parent != x then
    x.parent := find(x.parent)
  return x.parent
```

- This is called **path compression**.
Union-Find Analysis

- This is known as a “union-find” or “disjoint union” or “congruence closure” data structure.
- With the two optimizations, the amortized running time is $O(\text{inverse}\_\text{ackermann}(N))$
  - $\text{inverse}\_\text{ackermann}(N) \leq 5$ for $N \leq 2^{2^{65536}}$
- So the amortized time is effectively constant.
Equality Decision Procedure

Idea

• Idea: start with the graph based on the conjunction of literals given
  - $f(x) = g(z) \land x = y \land f(y) \neq g(z)$

• There is a unique minimal graph corresponding to the congruence closure of the quality relation (i.e., if you know $a = b$ and $b = c$, add the $a = c$ edge).

• Compute that graph via union-find, but when you learn $a = b$, also add $f(a) = f(b)$. 
Equality Decision Procedure
Intermediate Steps

let rec merge u v = (* add “u == v” *)
    if find(u) = find(v) then return ()
    let p_u = preds_of (eq_class_of u) in
    let p_v = preds_of (eq_class_of v) in
    union u v ;
    for each (x,y) in p_u, p_v do (* u=v => f(u) = f(v) *)
        if find x <> find y && congruent x y then
            merge x y
    let congruent x y =
        return (out_degree x = out_degree y)
        && forall i. find x.child[i] = find y.child[i]
Equality Decision Procedure

- **Input:**
  - \( t_1 = t'_1 \) \&\& \( t_2 = t'_2 \) \&\& ... \( t_p = t'_p \)
  - \( r_1 \neq r'_1 \) \&\& ... \( r_2 \neq r'_2 \) \&\& ... \( r_q \neq r'_q \)

- Construct the graph where the vertices correspond to the terms and the edges correspond to function application
  - for \( i = 1 \ldots p \): merge \( t_i t'_i \)
  - for \( i = 1 \ldots q \): if find \( r_i = \) find \( r'_i \) then FALSE

- TRUE
Equality Example

\[ g(g(g(x))) = x \quad \&\& \quad g(g(g(g(g(x)))))) = x \quad \&\& \quad g(x) \neq x \]

Input Term Graph

Union-Find Forest
Equality Example

\[ g(g(g(x))) = x \quad \&\& \quad g(g(g(g(g(x)))))) = x \quad \&\& \quad g(x) \neq x \]

Input Term Graph

Steps:
1. merge 4 1
2. merge 6 1
3. check find 2 1

Union-Find Forest
Equality Example

\[ g(g(g(x))) = x \land g(g(g(g(g(x))))) = x \land g(x) \neq x \]

**Steps:**
1. merge \(4\) \(1\)
2. merge \(6\) \(1\)
3. check find \(2\) \(1\)

\[ u = 4, \ v = 1 \]
\[ p_u = \{5,6\} \]
\[ p_v = \{2,3,4,5,6\} \]
merge \(u\) \(v\)
if congruent \(p_u\) \(p_v\)
merge \(p_u\) \(p_u\)

Input Term Graph

Union-Find Forest
Equality Example

\[ g(g(g(x))) = x \land g(g(g(g(g(x))))) = x \land g(x) \neq x \]

**Steps:**
1. merge 4 1
2. merge 6 1
3. check find 2 1

```

\begin{align*}
\text{u} &= 4, \text{v} = 1 \\
\text{p}_u &= \{5,6\} \\
\text{p}_v &= \{2,3,4,5,6\} \\
\text{merge } u \text{ v} \\
& \text{if congruent } p_u p_v \\
& \text{merge } p_u p_u
\end{align*}
```

**Input Term Graph**

**Union-Find Forest**
Equality Example

\[ g(g(g(x))) = x \quad \&\& \quad g(g(g(g(g(x))))) = x \quad \&\& \quad g(x) \neq x \]

**Input Term Graph**

**Steps:**
1. merge 4 1
2. merge 6 1
3. check find 2 1

**Union-Find Forest**

- \( u = 4, v = 1 \)
- \( p_u = \{5,6\} \)
- \( p_v = \{2,3,4,5,6\} \)

merge \( u \) \( v \)

\[ \text{if congruent 5 2} \]
\[ // 5 \neq 2, 5.\text{child} = 2.\text{child} \]

merge 5 2
Equality Example

\[ g(g(g(x))) = x \quad \&\& \quad g(g(g(g(g(x))))) = x \quad \&\& \quad g(x) \neq x \]

**Input Term Graph**

**Steps:**
1. merge 4 1
2. merge 6 1
3. check find 2 1

**Union-Find Forest**

\[ u = 4, v = 1 \]
\[ p_u = \{5,6\} \]
\[ p_v = \{2,3,4,5,6\} \]

merge \( u \) \( v \)

if congruent 5 2

// 5 != 2, 5.child = 2.child

merge 5 2
Equality Example

\[ g(g(g(x))) = x \quad \&\& \quad g(g(g(g(g(x)))))) = x \quad \&\& \quad g(x) \neq x \]

Steps:
1. merge 4 1
2. merge 6 1
3. check find 2 1

\[ u = 4, v = 1 \]
\[ p_u = \{5, 6\} \]
\[ p_v = \{2, 3, 4, 5, 6\} \]
merge \( u \) \( v \)

if congruent 6 3
// 6 != 3, 6.child = 3.child
merge 6 3

Input Term Graph

Union-Find Forest
Equality Example

\[ g(g(g(x))) = x \quad \text{and} \quad g(g(g(g(g(x))))) = x \quad \text{and} \quad g(x) \neq x \]

Steps:
1. merge 4 1
2. merge 6 1
3. check find 2 1

\[ u = 4, \quad v = 1 \]
\[ p_u = \{5, 6\} \]
\[ p_v = \{2, 3, 4, 5, 6\} \]
merge \( u \) \( v \)
if congruent 6 3
// 6 != 3, 6.child = 3.child
merge 6 3

Input Term Graph
Equality Example

\[ g(g(g(x))) = x \quad \&\& \quad g(g(g(g(g(x)))))) = x \quad \&\& \quad g(x) \neq x \]

Input Term Graph

Steps:
1. merge 4 1
2. merge 6 1
3. check find 2 1

\[ u = 6, \quad v = 1 \]
\[ p_u = \{4, 5, 6\} \]
\[ p_v = \{1, 2, 3, 4, 5, 6\} \]
merge \( u \) \( v \)
if congruent \( p_u \) \( p_v \)
mmerge \( p_u \) \( p_v \)

Union-Find Forest
Equality Example

\[ g(g(g(x))) = x \land g(g(g(g(g(x)))))) = x \land g(x) \neq x \]

Steps:
1. merge 4 1
2. merge 6 1
3. check find 2 1

\( u = 6, v = 1 \)
\( p_u = \{4, 5, 6\} \)
\( p_v = \{1, 2, 3, 4, 5, 6\} \)
merge \( u \) \( v \)
if congruent \( p_u \) \( p_v \)
merge \( p_u \) \( p_v \)

Union-Find Forest

Input Term Graph
Equality Example

g(g(g(x)))=x && g(g(g(g(g(x)))))=x && g(x)≠x

Input Term Graph

Steps:
1. merge 4 1
2. merge 6 1
3. check find 2 1

u = 6, v = 1
p_u = {4, 5, 6}
p_v = {1, 2, 3, 4, 5, 6}
merge u v
if congruent 4 2
// 4 != 2, 4.child = 2.child
merge 4 2

Union-Find Forest
Equality Example

\[ g(g(g(x))) = x \quad \text{&&} \quad g(g(g(g(g(x))))) = x \quad \text{&&} \quad g(x) \neq x \]

**Input Term Graph**

**Union-Find Forest**

Steps:
1. merge 4 1
2. merge 6 1
3. check find 2 1

\[ u = 6, \ v = 1 \]
\[ p_u = \{4,5,6\} \]
\[ p_v = \{1,2,3,4,5,6\} \]
merge \( u \) \( v \)
if congruent 4 2
// 4 \( \neq 2 \)
4.child = 2.child
merge 4 2
Equality Example

\[ g(g(g(x))) = x \quad \&\& \quad g(g(g(g(g(x))))) = x \quad \&\& \quad g(x) \neq x \]

Steps:
1. merge 4 1
2. merge 6 1
3. check find 2 1

\[ u = 6, \quad v = 1 \]
\[ p_u = \{4, 5, 6\} \]
\[ p_v = \{1, 2, 3, 4, 5, 6\} \]
merge u v
if congruent 4 2
// 4 \neq 2, 4.child = 2.child
merge 4 2
Equality Example

\[ g(g(g(x))) = x \quad \&\& \quad g(g(g(g(g(x)))))) = x \quad \&\& \quad g(x) \neq x \]

Input Term Graph

Union-Find Forest

Steps:
1. merge 4 1
2. merge 6 1
3. check find 2 1

they are equivalent: return false!
Q. Computer Science

• This algorithmic strategy is applicable to decomposable problems that exhibit the *optimal substructure property* (in which the optimal solution to a problem P can be constructed from the optimal solutions to its overlapping subproblems). The term was coined in the 1940's by Richard Bellman. Problems as diverse as “shortest path”, “sequence alignment” and “CFG parsing” use this approach.
Analogy: (NP-Hard) Reduction

Input Instance → Input Conversion \textit{(to match interface)} → Black-Box Oracle → Output Conversion → Final Answer
Theory Interface

- **Initialize**(universe : Literal Set)
- **SetTrue**(l : Literal) : Literal Set
  - Raise exception if l is inconsistent. Otherwise, add l to set of known facts. Return newly implied set of true facts (e.g., “a=c” after “a=b” and “b=c”)
- **Backtrack**(n : Nat)
  - Forget last n facts from “SetTrue”.
- **IsDefinitelyTrue**(l : Literal) : Bool
- **Explanation**(l : Literal) : Literal Set
  - If l is true, return a model (proof) of it.
Satisfiability Modulo Theories

- A **satisfiability modulo theories** (SMT) solver operates on propositions involving both logical terms and terms from theories.

- Modern SMT solvers can use any theory that satisfies the **Theory Interface** shown before.

- Replace Theory clauses with special propositional variables.

- Use a pure SAT solver. If the solution involves some theory clauses, ask the Theory if they can all be true. If not, **add constraints and restart**.
SMT Basic Idea

• Given a query like
  - \((x > 5) \land (p \lor (x < 4)) \land \neg p\)

• Note that almost everything can be handled by SAT:
  - \((\text{underline}\ x > 5) \land (p \lor (\text{underline}\ x < 4)) \land \neg p\)
  - Only the highlighted parts require a Theory.

• So ask SAT to consider:
  - \(T_1 \land (p \lor T_2) \land \neg p\)

• And then whenever SAT gives a model, ask the theories if that model makes sense.
SMT Architecture (Reduction Redux)

Input Instance

Mixed Constraints

Conversion via Mapping

Black-Box Oracle

DPLL SAT Solver

Theory:
Linear Arith equality, etc.

Black-Box Oracle

Model

Model

Output Conversion

Final Answer SAT

Final Answer UNSAT

Unsat

Sat

Sat

Unsat
SMT Example

- Input: \((x > 5) \land (p \lor (x < 4)) \land \neg p\)
- Rewrite: \(T_1 \land (p \lor T_2) \land \neg p\)
  - \(T_1 = “x > 5” \quad // mapping\)
  - \(T_2 = “x < 4”\)
- SAT solver returns \(\{T_1, T_2, \neg p\}\)
- Ask Theory about \(T_2 \land T_2\)
  - Theory Query: \((x > 5) \land (x < 4)\)
  - Theory Result: Unsatisfiable!
- \(T_1 \land (p \lor T_2) \land \neg p \land \neg (T_1 \land T_2)\)
Analogy: SMT vs. SLAM

- SLAM converted the *C program* to a *Boolean Program* in a sound manner:
  - If *Label* is reachable in the *Boolean Program* via a *Path P* and the ground truth (*Newton*) agrees with that *Path P*, the *Label* is truly reachable.
  - If *Newton* disagrees with that *Path P* because XYZ cannot be true in the *C program*, add XYZ to the abstraction and restart.

- SMT converts the *Mixed Constraints* into *Boolean Constraints* in a sound manner:
  - If the *Boolean Constraints* are *satisfiable* via a *Model M* and the ground truth (*Theories*) agree with that *Model M*, the constraints are truly *satisfiable*.
  - If the *Theories* disagree with *Model M* because XYZ cannot be true in the *Mixed Constraints*, add XYZ to the constraints and restart.
DPLL(T)

- **DPLL(T)** is an SMT algorithm based on the DPLL SAT solver but parameterized with respect to a set of Theories T.

- **DPLL(T) Algorithm**
  - Convert mixed constraints to boolean constraints
  - Run DPLL, but with two changes:
    - No Pure Variable Elimination optimization
    - Unit Propagation uses T.setTrue, may add clauses
  - Whenever T.setTrue raises an exception, add the negation of the explanation to the constraints

(homework #3 coding hint)
No Pure Variable Elimination

• In pure propositional logic, variables are necessarily independent.
  - So if P only appears positively, you can set P=true without loss and save time.

• With Theories, variables may be dependent.

• Consider:
  - \((x > 10 \lor x < 3) \land (x > 10 \lor x < 9) \land (x < 7)\)

• “\(x > 10\)” is always used positively
  - But just skipping to “\(x > 10\)” = true as part of the model leads you astray (makes the others false)!
Unit Propagation Additions

- Consider:
  - \((A = B) \land (B = C) \land (A \neq C \lor X)\)
- Convert:
  - \(P_1 \land P_2 \land (!P_3 \lor X)\)
- Unit Propagation: add \(P_1\) \((A = B)\) to model
- Now: \(P_2 \land (!P_3 \lor X)\)
- Unit Propagation: add \(P_2\) \((B = C)\) to model
  - But Wait! Theory reports: \(P_3\) \((A = C)\) now true
- Now: \(!P_3 \lor X\) \&\& \(P_3\) ...
DPLL(T) Example

- Input Clauses:
  - (f(a)!\equiv f(b) \lor d=e) && \quad // \ T1 = f(a) = f(b)
  - (a=b \lor a\neq c) && \quad // \ T2 = d=e
  - (a=b \lor c=b) && \quad // \ T3 = a=b
  - (a=c \lor X) && \quad // \ T4 = a=c
  - (c=b \lor X) \quad // \ T5 = c=b

- Converted:
  - (!T1 \lor !T2) (T3 \lor !T4) (T3 \lor T5) (T4 \lor X) (!T5 \lor !X)

- No Unit Clauses

- Heuristically choose T1 = False, add to model
DPLL(T) Example

- **Input Clauses:**
  - \((f(a)\neq f(b) \text{||} d\neq e) \&\& \quad // \ T1 = f(a)=f(b)\)
  - \((a=b \text{||} a\neq c) \&\& \quad // \ T2 = d=e\)
  - \((a=b \text{||} c=b) \&\& \quad // \ T3 = a=b\)
  - \((a=c \text{||} X) \&\& \quad // \ T4 = a=c\)
  - \((c\neq b \text{||} \neg X) \quad // \ T5 = c=b\)

- **Converted:**
  - \((\neg T1 \text{||} \neg T2) (T3 \text{||} \neg T4) (T3 \text{||} T5) (T4 \text{||} X) (\neg T5 \text{||} \neg X)\)

- No Unit Clauses

- Heuristically choose \(T1 = \text{False}\), add to model
  - Theory returns: \(\neg T3\)
DPLL(T) Example

• Input Clauses:
  - \( (f(a)\neq f(b) \quad \lor \quad d\neq e) \quad \land \quad \quad // \quad T1 = f(a) = f(b) \)
  - \( (a=b \quad \lor \quad a\neq c) \quad \land \quad \quad // \quad T2 = d = e \)
  - \( (a=b \quad \lor \quad c=b) \quad \land \quad \quad // \quad T3 = a = b \)
  - \( (a=c \quad \lor \quad X) \quad \land \quad \quad // \quad T4 = a = c \)
  - \( (c\neq b \quad \lor \quad \neg X) \quad \land \quad \quad // \quad T5 = c = b \)

• Converted:
  - \( (\neg T4) \quad (T5) \quad (T4 \quad \lor \quad X) \quad (\neg T5 \quad \lor \quad \neg X) \)

• Model: \( \neg T1, \neg T3 \)
DPLL(T) Example

- Input Clauses:
  - \((f(a)!=f(b) \; \text{||} \; d!=e) \; \&\&\) // T1 = f(a)=f(b)
  - \((a=b \; \text{||} \; a!=c) \; \&\&\) // T2 = d=e
  - \((a=b \; \text{||} \; c=b) \; \&\&\) // T3 = a=b
  - \((a=c \; \text{||} \; X) \; \&\&\) // T4 = a=c
  - \((c!=b \; \text{||} \; !X)\) // T5 = c=b

- Converted:
  - \((-T4) \; (T5) \; (T4 \; \text{||} \; X) \; (!T5 \; \text{||} \; !X)\)

- Model: !T1, !T3

- Unit Clauses: !T4, !T5, add to model
DPLL(T) Example

- **Input Clauses:**
  - (f(a)!=f(b) || d!=e) && // T1 = f(a)=f(b)
  - (a=b || a!=c) && // T2 = d=e
  - (a=b || c=b) && // T3 = a=b
  - (a=c || X) && // T4 = a=c
  - (c!=b || !X) // T5 = c=b

- **Converted:**
  - (X) (!X)

- **Model:** !T1, !T3, !T4, !T5
- **Unit Clause:** (X), add to model
DPLL(T) Example

• Input Clauses:
  - \((f(a)\neq f(b) \lor d\neq e)\) \&\& \quad // T1 = f(a) = f(b)
  - \((a=b \lor a\neq c)\) \&\& \quad // T2 = d = e
  - \((a=b \lor c=b)\) \&\& \quad // T3 = a = b
  - \((a=c \lor X)\) \&\& \quad // T4 = a = c
  - \((c\neq b \lor \neg X)\) \quad // T5 = c = b

• Converted:
  - \((X) \quad \neg(X)\)

• Model: \neg T1, \neg T3, \neg T4, \neg T5

• Unit Clause: \((X)\), add to model, contradiction!
  - Original input is not satisfiable!
DPLL(T) Conclusion

• DPLL(T) is widely used as a basis for modern SMT solving.
  - It is typically much faster than eagerly encoding all of the variables into bits (e.g., 32-bit integer: 32 boolean variables). It is general in that it allows many types of theories.

• For example, Microsoft's popular and powerful Z3 automated theorem prover handles many theories, but uses DPLL(T) + Simplex for linear inequalities.
  - “A Fast Linear-Arithmetic Solver for DPLL(T)”, 2006
Proofs

“Checking proofs ain’t like dustin’ crops, boy!”
Proof Generation

- We want our theorem prover to emit proofs
  - No need to trust the prover
  - Can find bugs in the prover
  - Can be used for proof-carrying code
  - Can be used to extract invariants
  - Can be used to extract models (e.g., in SLAM)

- Implements the soundness argument
  - On every run, a soundness proof is constructed
Proof Representation

- **Proofs are trees**
  - Leaves are hypotheses/axioms
  - Internal nodes are inference rules
- **Axiom: “true introduction”**
  - Constant: \textit{truei} : pf
  - \textit{pf} is the type of proofs
- **Inference: “conjunction introduction”**
  - Constant: \textit{andi} : pf → pf → pf
- **Inference: “conjunction elimination”**
  - Constant: \textit{andel} : pf → pf
- **Problem:**
  - “\textit{andel truei} : pf” but does not represent a valid proof
  - Need a powerful \textit{system that checks content}
Analogy: Integer Subtypes (cf. Hoare paper)

- We don't just need any number, we need a “number representing distance over time”
- WE don't just need any proof, we need a “proof representing $A \parallel B$"
Dependent Types, Preview

- Make \( pf \) a **family of types indexed by formulas**
  - \( f : Type \) (type of encodings of formulas)
  - \( e : Type \) (type of encodings of expressions)
  - \( pf : f \to Type \) (the type of proofs indexed by formulas: it is a proof *that f is true*)

- **Examples:**
  - \( \text{true} : f \)
  - \( \text{and} : f \to f \to f \)
  - \( \text{truei} : pf \text{ true} \)
  - \( \text{andi} : pf A \to pf B \to pf (\text{and} A B) \)
  - \( \text{andi} : \Pi A:f. \Pi B:f. \text{ pf } A \to \text{ pf } B \to \text{ pf } (\text{and} A B) \)
    - \( (\Pi A:f.X \text{ means “forall A of type f, dependent type } X”, \text{ see future lecture}) \)
Proof Checking

- Validate proof trees by **recursively checking** them
- Given a proof tree X claiming to prove A && B
- Must check X : pf (and A B)
- We use “**expression tree equality**”, so
  - andel (andi “1+2=3” “x=y”) does **not** have type pf (3=3)
  - This is already a proof system! If the proof-supplier wants to use the fact that 1+2=3 ⇔ 3=3, she can **include a proof of it** somewhere!
- Thus **Type Checking = Proof Checking**
  - And it’s quite easily **decidable**!
Proof Inference Rules

- What are some rules of inference and function types for:
  - Or introduction
    - Hint: or_introduction_left : pf A → pf (or A B)
  - Or elimination
  - Not introduction
  - Not elimination
  - Implies introduction
  - Implies elimination
  - False elimination
Bonus Question

• If we want to use Simplex to handle our Theory of Linear Inequalities, how do we handle ...
  - Equality \( x = 10 \)
  - Negative Literals \( !(x \leq 10) \)
  - Disequality \( x \neq 10 \)

• A **slack variable** converts an inequality to an equality.
  - Given: \( Ax \leq b \), create fresh \( y \geq 0 \)
  - Obtain: \( Ax + y = b \)
Homework

• HW2 Due Today

• Wes will be away Thursday and Friday
  - HW2 may not be graded before you are asked to turn in HW3. Sorry!

• HW3 Coding Hint:
  - You are only being asked to code up the “check this candidate model against the Theories and add new clauses if it doesn't work” part of SMT.