SMT: Satisfiability Modulo Theories

Ranjit Jhala, UC San Diego

April 9, 2013

Decision Procedures

Last Time

- Propositional Logic

Today

1. **Combining** SAT *and Theory* Solvers
2. **Theory Solvers**
    - Theory of *Equality*
    - Theory of *Uninterpreted Functions*
    - Theory of *Difference-Bounded Arithmetic*

Combining SAT and Theory Solvers

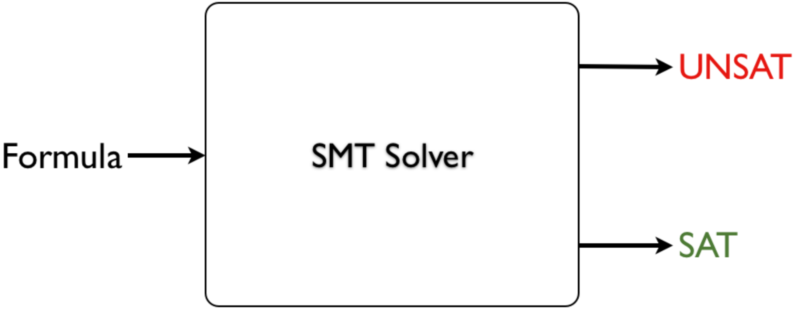

Figure: SMT Solver Architecture

Combining SAT and Theory Solvers

Goal Determine if a formula `f` is *Satisfiable*.

```
data Formula = Prop PVar          -- ^ Prop Logic
             | And  [Formula]     -- ^ ""
             | Or   [Formula]     -- ^ ""
             | Not  Formula       -- ^ ""
             | Atom Atom          -- ^ Theory Relation
```

Where theory elements are described by

```
data Expr    = Var TVar | Con Int | Op  Operator [Expr]

data Atom    = Rel Relation [Expr]
```

Split `Formula` into CNF + Theory Components

CNF Formulas

```
data Literal    = Pos PVar | Neg PVar
type Clause     = [Literal]
type CnfFormula = [Clause]
```

Split `Formula` into CNF + Theory Components

Theory Cube

A `TheoryCube` is an indexed list of `Atom`

```
data TheoryCube a  = [(a, Atom)]
```

Theory Formula

A `TheoryFormula` is a `TheoryCube` indexed by `Literal`

```
type TheoryFormula = TheoryCube Literal
```

- **Conjunction** of **assignments** of each literal to theory `Atom`

Split `Formula` into CNF + Theory Components

Split SMT Formulas

An `SmtFormula` is a pair of `CnfFormula` and `TheoryFormula`

```
type SmtFormula    = (CnfFormula, TheoryFormula)
```

Theorem There is a *poly-time* function

```
toSmt :: Formula -> SmtFormula
toSmt = error "Exercise For The Reader"
```

Split `SmtFormula` : Example

Consider the formula

- $(a = b \vee a = c) \wedge (b = d \vee b = e) \wedge (c = d) \wedge (a \neq d) \wedge (a \neq e)$

We can split it into **CNF**

- $(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5) \wedge (x_6) \wedge (x_7)$

And a **Theory Cube**

- $(x_1 \leftrightarrow a = b), (x_2 \leftrightarrow a = c), (x_3 \leftrightarrow b = d), (x_4 \leftrightarrow b = e)$
  $(x_5 \leftrightarrow c = d), (x_6 \leftrightarrow a \neq d), (x_7 \leftrightarrow a \neq e)$

Split `SmtFormula` : Example

Consider the formula

- $(a = b \vee a = c) \wedge (b = d \vee b = e) \wedge (c = d) \wedge (a \neq d) \wedge (a \neq e)$

We can split it into a `CnfFormula`

```
( [[1, 2], [3, 4], [5], [6], [7]]
```

and a `TheoryFormula`

```
[ (1, Rel Eq ["a", "b"]), (2, Rel Eq ["a", "c"])
, (3, Rel Eq ["b", "d"]), (4, Rel Eq ["b", "e"])
, (5, Rel Eq ["c", "d"])
, (6, Rel Ne ["a", "d"]), (7, Rel Ne ["a", "e"]) ]
```

Combining SAT and Theory Solvers: Architecture

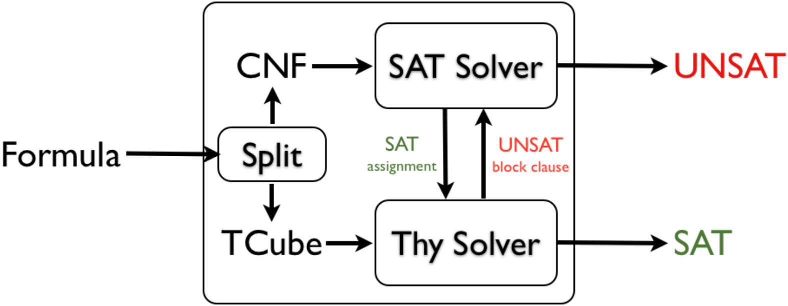

Figure: SMT Solver Architecture

Combining SAT and Theory Solvers: Architecture

Lets see this in code

```
smtSolver :: Formula -> Result
smtSolver = smtLoop . toSmt
```

Combining SAT and Theory Solvers: Architecture

Lets see this in code

```
smtLoop   :: SmtFormula -> Result
smtLoop (cnf, thy) =
  case satSolver cnf of
    UNSAT -> UNSAT
    SAT s -> case theorySolver $ cube thy s of
               SAT     -> SAT
               UNSAT c -> smtLoop (c:cnf) thy
```

Where, the function

```
cube :: TheoryFormula -> [Literal] -> TheoryFormula
```

Returns a **conjunction of atoms** for the `theorySolver`

Combining SAT and Theory Solvers: Architecture

Lets see this in code

```
smtLoop   :: SmtFormula -> Result
smtLoop (cnf, thy) =
  case satSolver cnf of
    UNSAT -> UNSAT
    SAT s -> case theorySolver $ cube thy s of
               SAT     -> SAT
               UNSAT c -> smtLoop (c:cnf) thy
```

In `UNSAT` case `theorySolver` returns **blocking clause**

- Tells `satSolver` not to find *similar* assignments ever again!

`smtSolver` : Example

Recall formula split into **CNF**

- $(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5) \wedge (x_6) \wedge (x_7)$

and **Theory Cube** -
$(x_1 \leftrightarrow a = b), (x_2 \leftrightarrow a = c), (x_3 \leftrightarrow b = d), (x_4 \leftrightarrow b = e)$
$(x_5 \leftrightarrow c = d), (x_6 \leftrightarrow a \neq d), (x_7 \leftrightarrow a \neq e)$

Iteration 1: SAT

- **In** $(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5) \wedge (x_6) \wedge (x_7)$
- **Out** `SAT` $x_1 \wedge x_3 \wedge x_5 \wedge x_6 \wedge x_7$

Iteration 1: SMT

- **In** $(x_1, a = b), (x_3, b = d), (x_5, c = d), (x_6, a \neq d), (x_7, a \neq e)$
- **Out** `UNSAT` $(\neg x_1 \vee \neg x_3 \vee \neg x_6)$

`smtSolver` : Example

Iteration 2: SAT

- **In** $(x_1 \vee x_2), (x_3 \vee x_4), (x_5), (x_6), (x_7), (\neg x_1 \vee \neg x_3)$
- **Out** `SAT` $x_1 \wedge x_4 \wedge x_5 \wedge x_6 \wedge x_7$

Iteration 2: SMT

- **In** $(x_1, a = b), (x_4, b = e), (x_5, c = d), (x_6, a \neq d), (x_7, a \neq e)$
- **Out** `UNSAT` $(\neg x_1 \vee \neg x_4 \vee \neg x_7)$

`smtSolver` : Example

Iteration 3 : SAT

- **In** $(x_1 \vee x_2), (x_3 \vee x_4), (x_5), (x_6), (x_7)$, $(\neg x_1 \vee \neg x_3), (\neg x_1 \vee \neg x_4 \vee \neg x_7)$
- **Out** `SAT` $x_2 \wedge x_4 \wedge x_5 \wedge x_6 \wedge x_7$

Iteration 3 : SMT

- **In** $(x_2, a = c), (x_4, b = e), (x_5, c = d), (x_6, a \neq d), (x_7, a \neq e)$
- **Out** `UNSAT` $(\neg x_2 \vee \neg x_5 \vee \neg x_6)$

smtSolver : Example

Iteration 4 : SAT

- **In** $(x_1 \vee x_2), (x_3 \vee x_4), (x_5), (x_6), (x_7),$
  $(\neg x_1 \vee \neg x_3), (\neg x_1 \vee \neg x_4 \vee \neg x_7),\ (\neg x_2 \vee \neg x_5 \vee \neg x_6)$
- **Out** UNSAT
- Thus smtSolver returns UNSAT

Today

1. Combining SAT *and Theory* Solvers
2. **Theory Solvers**
   - Theory of *Equality*
   - Theory of *Uninterpreted Functions*
   - Theory of *Difference-Bounded Arithmetic*

Issue: How to solve formulas over *different* theories?

Need to Solve Formulas Over Different Theories

Input formulas `F` have `Relation`, `Operator` from *different* theories

- $F \equiv f(f(a) - f(b)) \neq f(c), b \geq a, c \geq b + c, c \geq 0$
- Recall here *comma* means *conjunction*

Formula contains symbols from

- `EUF` : $f(a)$, $f(b)$, $=$, $\neq$,...
- `Arith` : $\geq$, $+$, 0,...

How to solve formulas over *different* theories?

Naive Splitting Approach

Consider F over T_E (e.g. `EUF`) and T_A (e.g. `Arith`)

By Theory, Split F Into $F_E \wedge F_A$

- F_E which **only contains** symbols from T_E
- F_A which **only contains** symbols from T_A

Our example,

- $F \equiv f(f(a) - f(b)) \neq f(c), b \geq a, c \geq b + c, c \geq 0$

Can be split into

- $F_E \equiv f(f(a) - f(b)) \neq f(c)$
- $F_A \equiv b \geq a, c \geq b + c, c \geq 0$

Naive Splitting Approach

Our example,

- $F \equiv f(f(a) - f(b)) \neq f(c), b \geq a, c \geq b + c, c \geq 0$

Can be split into

- $F_E \equiv f(f(a) - f(b)) \neq f(c)$
- $F_A \equiv b \geq a, c \geq b + c, c \geq 0$

Problem! Pesky "minus" operator ($-$) has crept into F_E ...

Less Naive Splitting Approach

Problem! Pesky "minus" operator ($-$) has crept into F_E ...

Purify Sub-Expressions With Fresh Variables

- Replace $r(f(e)$ with $t = f(e) \wedge r(t)$
- So that each *atom* belongs to a *single* theory

Example formula F becomes

- $t_1 = f(a), t_2 = f(b), t_3 = t_1 - t_2$
- $f(t3) \neq f(c), b \geq a, c \geq b + c, c \geq 0$

Which splits nicely into

- $F_E \equiv t_1 = f(a), t_2 = f(b), f(t3) \neq f(c)$
- $F_A \equiv t_3 = t_1 - t_2, b \geq a, c \geq b + c, c \geq 0$

Less Naive Splitting Approach

Consider F over T_E (e.g. EUF) and T_A (e.g. Arith)

- **Split** $F \equiv F_E \wedge F_A$

Now what? Run theory solvers independently

```
theorySolver f =
  let (fE, fA) = splitByTheory f in
  case theorySolverE fE, theorySolverA fA of
    (UNSAT, _) -> UNSAT
    (_, UNSAT) -> UNSAT
    (SAT, SAT) -> SAT
```

Will it work?

Less Naive Splitting Approach

Run Theory Solvers Independently

```
theorySolver f =
  let (fE, fA) = splitByTheory f in
  case theorySolverE fE, theorySolverA fA of
    (UNSAT, _) -> UNSAT
    (_, UNSAT) -> UNSAT
    (SAT, SAT) -> SAT
```

Will it work? Alas, no.

Satisfiability of Mixed Theories

Consider F over T_E (e.g. `EUF`) and T_A (e.g. `Arith`)

- **Split** $F \equiv F_E \wedge F_A$

The following are obvious

1. $UNSAT\ F_E$ implies $UNSAT\ F_E \wedge F_A$ implies $UNSAT\ F$
2. $UNSAT\ F_A$ implies $UNSAT\ F_E \wedge F_A$ implies $UNSAT\ F$

But this **is not true**

3. $SAT\ F_E$ and *SAT F_A implies $SAT\ F_E \wedge F_A$

Satisfiability of Mixed Theories

$SAT\ F_E$ and $SAT\ F_A$ **does not imply** $SAT\ F_E \wedge F_A$

Example

- $F_E \equiv t_1 = f(a), t_2 = f(b), f(t3) \neq f(c)$
- $F_A \equiv t_3 = t_1 - t_2, b \geq a, c \geq b + c, c \geq 0$

Individual Satisfying Assignment

- Let $\sigma \equiv = a \mapsto 0, b \mapsto 0, c \mapsto 1, f \mapsto \lambda x.x$
- Easy to check that σ satisfies F_E and F_A
- (But not both!)

One bad assignment doesn't mean F is *UNSAT*...

Proof of Unsatisfiability of Mixed Formula $F_E \wedge F_A$

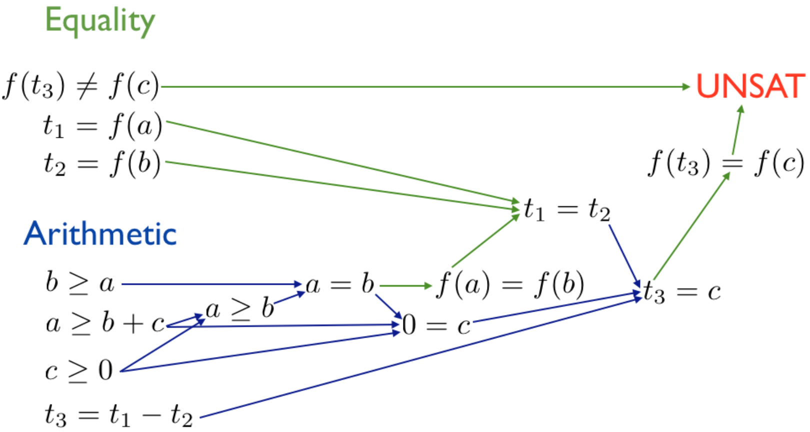

Figure: Proof Of Unsatisfiability

Satisfiability of Mixed Theories

Is quite non-trivial!

- EUF: Ackermann, 1954
- Arith: Fourier, 1827
- EUF+Arith: Nelson-Oppen, POPL 1978

Real software verification queries span multiple theories

- EUF + Arith + Arrays + Bit-Vectors + . . .

Good news! The Nelson - Oppen *combination* procedure . . .

Nelson-Oppen Framework For Combining Theory Solvers

Step 1

- **Purify** each atom with fresh variables
- **Result** each `Atom` belongs to *one* theory

Step 2

- **Check Satisfiability** of each theory using its solver
- **Result** If *any* solver says `UNSAT` then formula is `UNSAT`

Step 3 (Key Insight)

- **Broadcast New Equalities** discovered by *each* solver
- **Repeat** step 2 **until** no new equalities discovered

Nelson-Oppen Framework: Example

Input

- $F \equiv f(f(a) - f(b)) \neq f(c), b \geq a, c \geq b + c, c \geq 0$

After Step 1 (Purify)

- $t_1 = f(a), t_2 = f(b), t_3 = t_1 - t_2$
- $f(t3) \neq f(c), b \geq a, c \geq b + c, c \geq 0$

Nelson-Oppen Framework: Example

After Step 2 (Run `EUF` on F_E, `Arith` on F_A)

- $F_E \equiv t_1 = f(a), t_2 = f(b), f(t3) \neq f(c)$ is `SAT`
- $F_A \equiv t_3 = t_1 - t_2, b \geq a, c \geq b + c, c \geq 0$ is `SAT`

After Step 3

- `Arith` *discovers* $a = b$

Broadcast

- $F'_E \leftarrow F_E, a = b$

Repeat Step 2

Nelson-Oppen Framework: Example

After Step 2 (Run `EUF` on F'_E, `Arith` on F_A)

- $F'_E \equiv t_1 = f(a), t_2 = f(b), f(t3) \neq f(c), a = b$ is `SAT`
- $F_A \equiv t_3 = t_1 - t_2, b \geq a, c \geq b + c, c \geq 0$ is `SAT`

After Step 3

- `EUF` *discovers* $t_1 = t_2$

Broadcast and Update

- $F'_A \leftarrow F_A, t_1 = t_2$

Repeat Step 2

Nelson-Oppen Framework: Example

After Step 2 (Run `EUF` on F'_E, `Arith` on F'_A)

- $F'_E \equiv t_1 = f(a), t_2 = f(b), f(t3) \neq f(c), a = b$ is `SAT`
- $F'_A \equiv t_3 = t_1 - t_2, b \geq a, c \geq b + c, c \geq 0, t_1 = t_2$ is `SAT`

After Step 3

- `Arith` *discovers* $t_3 = c$

Broadcast and Update

- $F''_E \leftarrow F'_E, t_3 = c$

Repeat Step 2

Nelson-Oppen Framework: Example

After Step 2 (Run `EUF` on F''_E, `Arith` on F'_A)

- $F'_E \equiv t_1 = f(a), t_2 = f(b), f(t3) \neq f(c), a = b, t3 = c$
- `Arith` returns `UNSAT`
- **Output** `UNSAT`

Nelson-Oppen in Code

TODO

Nelson-Oppen Framework For Combining Theory Solvers

A Theory T is Stably Infinite

If every T-satisfiable formula has an infinite model

- Roughly, is SAT over a universe with infinitely many *Values*

A Theory T is Convex

If whenever F implies $a_1 = b_1 \vee a_2 = b_2$

either F implies $a_1 = b_1$ **or** F implies $a_2 = b_2$

Nelson-Oppen Framework For Combining Theory Solvers

Theorem: Nelson-Oppen Combination

Let T_1, T_2 be *stably infinite*, *convex* theories w/ solvers `S1` and `S2`

1. `nelsonOppen S1 S2` is a solver the combined theory $T_1 \cup T_2$
2. `nelsonOppen S1 S2 F == SAT` iff `F` is satisfiable in $T_1 \cup T_2$.

Convexity

The **convexity** requirement is the important one in practice.

Example of Non-Convex Theory

$(\mathbb{Z}, +, \leq)$ and Equality

- $F \equiv 1 \leq a \leq 2, b = 1, c = 2, t_1 = f(a), t_2 = f(b), t_3 = f(c)$
- F implies $t_1 = t_2 \vee t_1 = t_3$
- F does not imply either $t_1 = t_2$ or $t_1 = t_3$

Nelson-Oppen **fails** on $F, t_1 \neq t_2, t_1 \neq t_3$

- Extensions: add case-splits on dis/equality ## Nelson-Oppen Architecture

TODO Nifty Bus PIC

What is the **API** for each Theory Solver?

Requirements of Theory Solvers

Recall the `smtLoop` architecture

```
smtLoop   :: SmtFormula -> Result
smtLoop (cnf, thy) =
  case satSolver cnf of
    UNSAT -> UNSAT
    SAT s -> case theorySolver $ cube thy s of
               SAT     -> SAT
               UNSAT c -> smtLoop (c:cnf) thy
```

Requirement of `theorySolver`

- `SAT` : Each solver broadcast equalities
- `UNSAT` : Each solver broadcast **cause** of equalities
- `theorySolver` constructs **blocking clause** from *causes*

Building Blocking Clauses from Causes

- **Tag** each input `Atom`
- **Tag** each discovered and broadcasted equality
- **Link** each discovered fact with *tags* of its causes
- On **UNSAT** returned cause is backwards *slice* of *tags*
- Will see this informally, but will show up in assignment. . .

Today

1. Combining SAT *and Theory* Solvers
2. **Combining Solvers for Multiple Theories**
   - **Theory of Equality**
   - Theory of *Uninterpreted Functions*
   - Theory of *Difference-Bounded Arithmetic*

Solver for Theory of Equality

Recall Only need to solve list of `Atom`

- i.e. formulas like $\bigwedge_{i,j} e_i = e_j \wedge \bigwedge_{k,l} e_k \neq e_l$

Axioms for Theory of Equality

Rules defining when one expressions *is equal to* another.

Reflexivity: Every term e is equal to itself

$$\forall e. e = e$$

Symmetry: If e_1 is equal to e_2, then e_2 is equal to e_1

$$\forall e_1, e_2. \text{If } e_1 = e_2 \text{ Then } e_2 = e_1$$

Transitivity: If e_1 equals e_2 and e_2 equals e_3 then e_1 equals e_3

$$\forall e_1, e_2, e_3. \text{If } e_1 = e_2 \text{ and } e_2 = e_3 \text{ Then } e_1 = e_3$$

Solver for Theory of Equality

Let R be a relation on expressions.

Equivalence Closure of R

Is the *smallest* relation containing R that is *closed* under

- Reflexivity
- Symmetry
- Transitivity

By definition, closure is an *equivalence* relation

Solver: Compute Equivalence Closure of Input Equalities

- Compute equivalence closure of input equality atoms
- Return `UNSAT` if any disequal terms are in the closure
- Return `SAT` otherwise

Solver for Theory of Equality

Input $\bigwedge_{i,j} e_i = e_j \wedge \bigwedge_{k,l} e_k \neq e_l$

Step 1 Build Undirected Graph

- *Vertices* $e_1, e_2, \ldots$
- *Edges* $e_i - - - e_j$ for each equality atom $e_i = e_j$

Step 2 Compute Equivalence Closure

- Add edges between e and e' per *transitivity* axioms

Note: Reflex. and Symm. handled by graph representation

Output For each k, l in disequality atoms,

- If exists edge $e_k - - - e_l$ in graph then return `UNSAT`
- Else return `SAT`

Solver for Theory of Equality: Example

Input formula: $a = b, b = d, c = e, a \neq d, a \neq e$

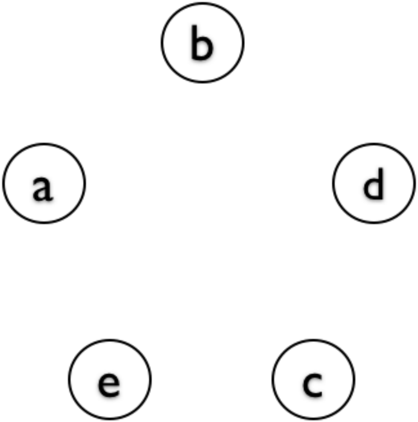

Figure: Inital Graph: Vertices

Solver for Theory of Equality: Example

Input formula: $a = b, b = d, c = e, a \neq d, a \neq e$

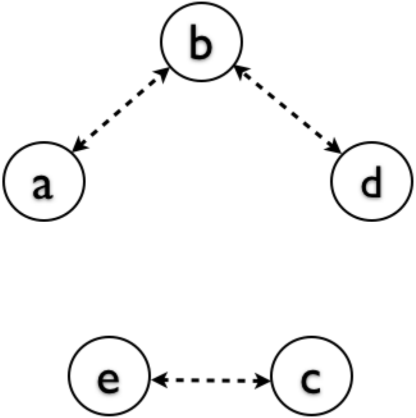

Figure: Inital Graph: Edges From Atoms

Solver for Theory of Equality: Example

Input formula: $a = b, b = d, c = e, a \neq d, a \not e$

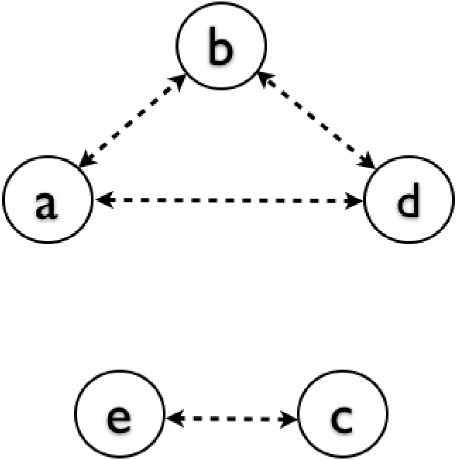

Figure: Inital Graph: Equivalence Closure

Solver for Theory of Equality: Example

Input formula: $a = b, b = d, c = e, a \neq d, a \not e$

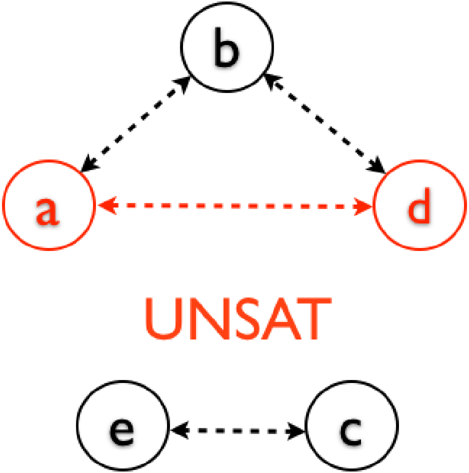

Figure: Inital Graph: Check Disequalities

Solver for Theory of Equality

That was a **slow** algorithm

- Worst case number of edges is quadratic in number of expressions

Better approach using **Union-Find**

Solver for Theory of Equality: Union-Find Algorithm

Key Idea

- Build **directed tree** of nodes for each equivalent set
- Tree root is **canonical representative** of equivalent set
- i.e. nodes are equal *iff* they have the **same root**

`find e`

- Walks up the tree and returns the **root** of `e`

`union e1 e2`

- Updates graph with equality `e1 == e2`
- Merges equivalence sets of `e1` and `e2`

```
union e1 e2 = do r1 <- find e1
                 r2 <- find e2
                 link r1 r2
```

Union Find : Example

Graph represents fact that $a = b = c = d$ and $e = f = g$.

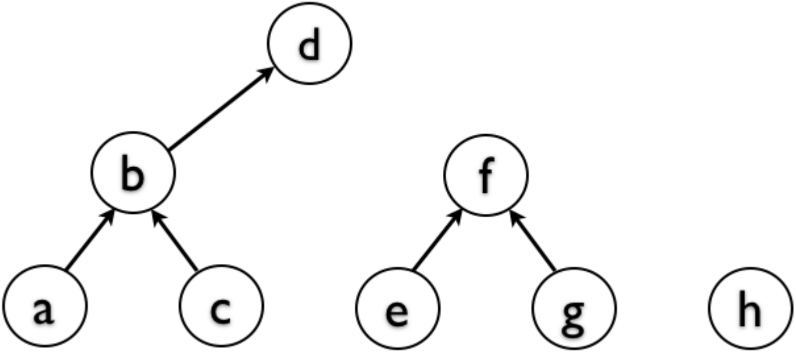

Figure: Inital Union-Find Graph

Union-Find : Example

Graph represents fact that $a = b = c = d$ and $e = f = g$.

Updates graph with equality $a = e$ using `union a e`

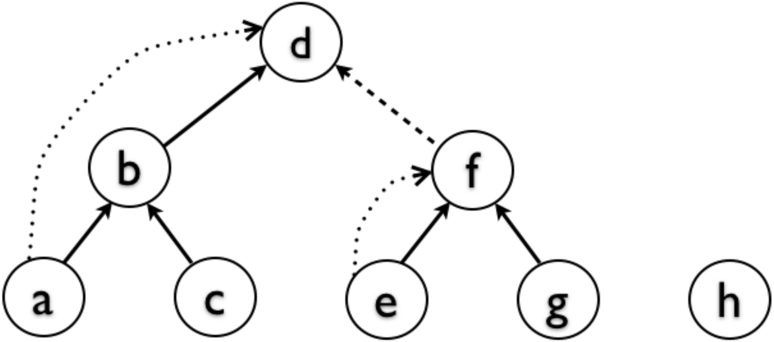

Figure: Find Roots of a and e

Union-Find : Example

After linking, graph represents fact that
$a = b = c = d = e = f = g$.

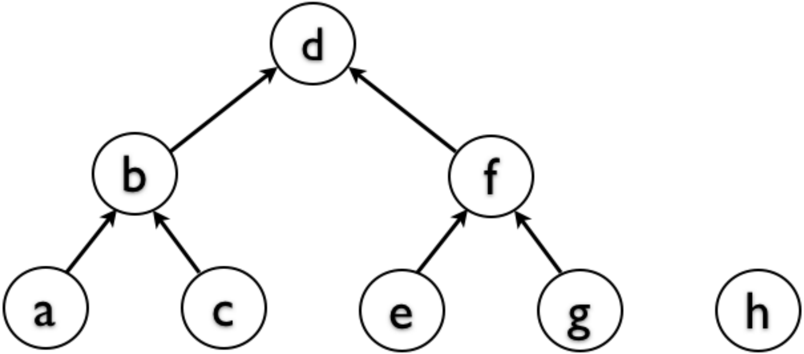

Figure: Union The Sets of a and e

Solver for Theory of Equality: Union-Find Algorithm

Algorithm

```
theorySolverEq atoms
  = do _   <- forM_ eqs  union          -- 1. Build U-F Tree
       u   <- anyM  neqs checkEqual -- 2. Check Conflict
       return $ if u then UNSAT else SAT
    where
      eqs     = [(e, e') | (e `Eq` e') <- atoms]
      neqs    = [(e, e') | (e `Ne` e') <- atoms]

checkEqual (e, e')
  = do r     <- find e
       r'    <- find e'
       return $ r == r'
```

Solver for Theory of Equality: Missing Pieces

1. How to **discover equalities** ?
2. How to **track causes** ?

Figure it out in *homework*

Today

1. Combining SAT *and Theory* Solvers
2. Combining Solvers for multiple theories
   - Theory of Equality
   - **Theory of Uninterpreted Functions**
   - Theory of *Difference-Bounded Arithmetic*

Solver for Theory of Equality + Uninterpreted Functions

Recall Only need to solve list of `Atom`

- i.e. formulas like $\bigwedge_{i,j} e_i = e_j \wedge \bigwedge_{k,l} e_k \neq e_l$

New: UIF Applications in Expressions

- An expression e can be of the form $f(e_1, \ldots, e_k)$
- Where f is an *uninterpreted function* of arity k

Question: What does *uninterpreted* mean anyway ?

Axioms for Theory of Equality + Uninterpreted Functions

Rules defining when one expressions *is equal to* another.

Equivalence Axioms

- Reflexivity
- Symmetry
- Transitivity

Congruence

If function arguments are equal, then outputs are equal

$$\forall e_i, e'_i.\ \text{If}\ \wedge_i\ e_i = e'_i\ \text{Then}\ f(e_1, \ldots, e_k) = f(e'_1, \ldots, e'_k)$$

Solver for Theory of Equality + Uninterpreted Functions

Let R be a relation on expressions.

Congruence Closure of R

Is the *smallest* relation containing R that is *closed* under

- Reflexivity
- Symmetry
- Transitivity
- Congruence

Solver: Compute Congruence Closure of Input Equalities

- Compute **congruence closure** of input equality atoms
- Return `UNSAT` if any *disequal* terms are in the closure
- Return `SAT` otherwise

Solver for EUF: Extended Union-Find Algorithm

Step 1: Represent Expressions With DAG

- Each DAG **node** implicit **fresh variable** for sub-expression
- Shared across theory solvers

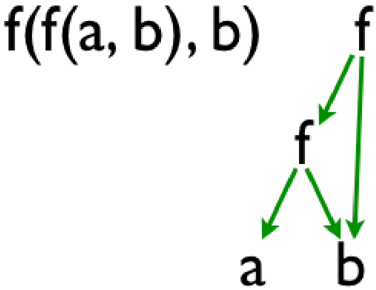

Figure: DAG Representation of Expressions

Solver for EUF: Extended Union-Find Algorithm

Step 2: Keep Parent Links to Function Symbols

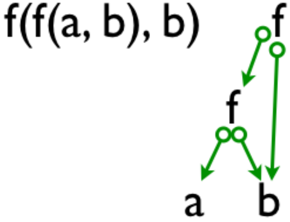

Figure: Parent Links

Solver for EUF: Extended Union-Find Algorithm

Step 3: Extend `union e1 e2` To Parents

```
union e1 e2
  = do e1' <- find e1
       e2' <- find e2
       link        e1' e2'
       linkParents e1' e2'

linkParents e1' e2'
  = do transferParents       e1' e2'
       recursiveParentUnion e1' e2'
```

Solver for EUF: Example

Input $a = f(f(f(a))), a = f(f(f(f(f(a)), x \neq f(a)$

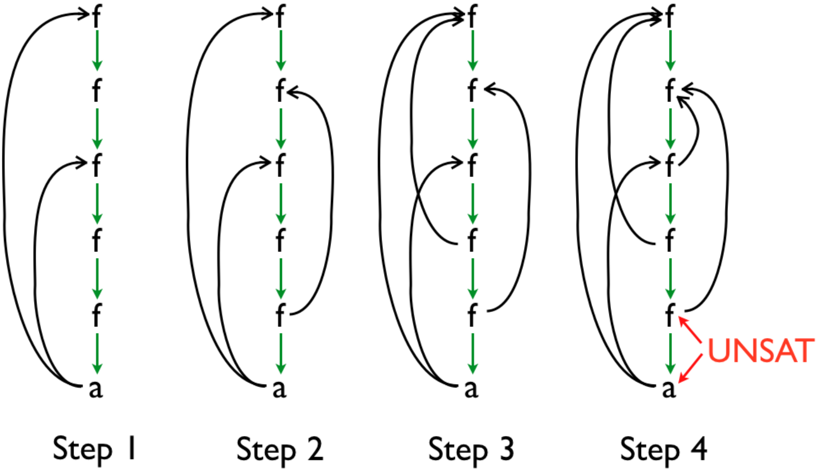

Figure: Congruence Closure Example

Solver for Theory of EUF: Missing Pieces

1. How to **discover equalities** ?
2. How to **track causes** ?

Figure it out in *homework*

Today

1. Combining SAT *and Theory* Solvers
2. Combining Solvers for multiple theories
   - Theory of Equality
   - Theory of Uninterpreted Functions
   - **Theory of Difference-Bounded Arithmetic**

Theory of Linear Arithmetic

- Operators $+$, $-$, $=$, $<$, 0, 1, -1, 2, -2, ...
- Semantics: as expected
- The most useful in program verification after equality
- Example: $b > 2a + 1$, $a + b > 1$, $b < 0$

Decision Procedure:

- Linear Programming / e.g. Simplex (Over Rationals)
- Integer Linear Programming (Over Integers)

Theory of Difference Constraints

Special case of linear arithmetic, with atoms

$$a - b \leq n$$

where a, b are variables, n is constant integer.

Can express many common linear constraints

Special variable z representing 0

- $a = b \equiv a - b \leq 0,\ b - a \leq 0$
- $a \leq n \equiv a - z \leq n$
- $a \geq n \equiv z - a \leq -n$
- $a < b \equiv a - b \leq -1$
- *etc.*

Solver For Difference Constraints

How to check satisfiability?

Directed Graph Based Procedure

Vertices for each *variable*

Edges for each *constraint*

Example: Atoms

- $a - b \leq 0$
- $b - c \leq -4$
- $c - a \leq 2$
- $c - d \leq -1$

Algorithm

TODO

Solver For Difference Constraints

Theorem: A set of difference constraints is satisfiable iff there is no **negative weight** cycle in the graph.

- Can be solved in $O(V.E)$ Bellman-Ford Algorithm
- $V =$ number of vertices
- $E =$ number of edges

Issues

1. Why does it work?
2. How to detect equalities?
3. How to track causes?

Today

1. Combining SAT *and Theory* Solvers
2. Combining Solvers for multiple theories
   - Theory of Equality
   - Theory of Uninterpreted Functions
   - Theory of Difference-Bounded Arithmetic
3. **Other Theories**
   - Lists
   - Arrays
   - Sets
   - Bitvectors
   - ...