SMT Solving
A Core Theory Solver

CS560: Reasoning About Programs

Roopsha Samanta

Partly based on slides by Isil Dillig and Emina Torlak
Roadmap

Previously

- Propositional logic and SAT solving
- First-order logic and first-order theories

Today

- SMT solving
- DPLL(T) : Combine DPLL algorithm for SAT solving with theory solvers
- A core theory solver: congruence closure algorithm for $T_=$
SMT solver

- Planning and scheduling in AI
- Network analysis
- Test-case generation
- Software verification
- White-box fuzzing for security

::
Planning and scheduling in AI

Network analysis

Test-case generation

Software verification

White-box fuzzing for security

SMT solver

- Z3 (MSR)
- Yices (SRI)
- MathSAT (U Trento)
- CVC3 (NYU, U Iowa)
- STP (Stanford)
- Barcelogic (Catalunya)

...
Planning and scheduling in AI
Network analysis
Test-case generation
Software verification
White-box fuzzing for security

SMT solver

Z3 (MSR)
Yices (SRI)
MathSAT (U Trento)
CVC3 (NYU, U Iowa)
STP (Stanford)
Barcelogic (Catalunya)

SMT-Lib: library of benchmarks
http://www.smtlib.org

SMT-Comp: annual SMT-Solver competition
http://www.smtcomp.org
\[
x = f(y)
\]
\[
2x + z < 10
\]
\[
3y = 1
\]
\[
\vdots
\]
\[
a[i] = x
\]

SMT solver
\begin{align*}
x &= f(y) \\
2x + z &< 10 \\
3y &= 1 \\
\vdots \\
a[i] &= x
\end{align*}

SMT solver

Theories
\[ x = f(y) \]
\[ 2x + z < 10 \]
\[ 3y = 1 \]
\[ \vdots \]
\[ a[i] = x \]

Theories

First-order logic

SMT solver
Theories

First-order logic

\[ x = f(y) \]
\[ 2x + z < 10 \]
\[ 3y = 1 \]
\[ \vdots \]
\[ a[i] = x \]

Core solver

DPLL(\(T\))

Theory solver

Theory solver

Theory equation here.
\[ x = f(y) \]
\[ 2x + z < 10 \]
\[ 3y = 1 \]
\[ \vdots \]
\[ a[i] = x \]

Decision procedure for checking satisfiability in quantifier-free conjunctive fragment

Theories

First-order logic

\[ x = f(y) \]
\[ 2x + z < 10 \]
\[ 3y = 1 \]
\[ \vdots \]
\[ a[i] = x \]

SMT solver

Core solver

DPLL(\(T\))

Theory solver

…

Theory solver
DPLL($T$): Main Idea

**Boolean abstraction** of SMT formula:
Treat each atomic formula as a propositional variable
DPLL(\(T\)): Main Idea

Boolean abstraction of SMT formula:
Treat each atomic formula as a propositional variable

SAT solver handles Boolean structure of formula:
- If there is no satisfying assignment to Boolean abstraction, SMT formula is UNSAT
- If there is satisfying assignment to Boolean abstraction, SMT formula may not be SAT

\[ F : x=2 \land ((y=2 \land x<2) \lor (x=2)) \]

\[ B(F) : b_1 \land (b_2 \land b_3) \lor \neg b_1 \]

\[ A : b_1 \land b_2 \land b_3 \]

\[ B'(A) = x=2 \land y=2 \land x<2 \]
DPLL(T): Main Idea

Boolean abstraction of SMT formula:
Treat each atomic formula as a propositional variable

SAT solver handles Boolean structure of formula
- If there is no satisfying assignment to Boolean abstraction, SMT formula is UNSAT
- If there is satisfying assignment to Boolean abstraction, SMT formula may not be SAT

Theory solver checks whether assignment made by SAT solver is satisfiable modulo theory
**DPLL(\(T\)): Main Idea**

**Boolean abstraction** of SMT formula:
Treat each atomic formula as a propositional variable

SAT solver handles Boolean structure of formula
- If there is no satisfying assignment to Boolean abstraction, SMT formula is UNSAT
- If there is satisfying assignment to Boolean abstraction, SMT formula may not be SAT

Theory solver checks whether assignment made by SAT solver is satisfiable modulo theory

If SAT solver finds assignment that is consistent with theory, then SMT formula is satisfiable
SMT Formulas and Boolean Abstraction

- SMT formula in theory $T$:
  \[ F := a_T^i \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \neg F \]
  
- For each SMT formula, define a bijective function $\mathcal{B}$, called **Boolean abstraction function** (or Boolean skeleton), that maps SMT formula to an overapproximate SAT formula.

- Function $\mathcal{B}$ defined inductively as follows:
  \[
  \begin{align*}
  \mathcal{B}(a_T^i) &= b_i \\
  \mathcal{B}(F_1 \land F_2) &= \mathcal{B}(F_1) \land \mathcal{B}(F_2) \\
  \mathcal{B}(F_1 \lor F_2) &= \mathcal{B}(F_1) \lor \mathcal{B}(F_2) \\
  \mathcal{B}(\neg F) &= \neg \mathcal{B}(F)
  \end{align*}
  \]
DPLL($T$)

\[ DPLL_T(F) \]
\[
G = B(F) \\
\text{while (true) do} \\
\quad A, out = \text{SAT-SOLVER}(G) \\
\quad \text{if (out = UNSAT) then return UNSAT} \\
\quad \text{else} \\
\quad \quad out = T\text{-SOLVER}(B^{-1}(A)) \\
\quad \quad \text{if (out = SAT) then return SAT} \\
\quad \quad \text{else } G = G \land \neg A
\]
DPLL(T)

DPLL_{\tau}(F)
\[ G = \mathcal{B}(F) \]

While (true) do

\[ A, \text{out} = \text{SAT-SOLVER}(G) \]

If (out = UNSAT) then return UNSAT

Else

out = T-SOLVER (\mathcal{B}^{-1}(A))

If (out = SAT) then return SAT

Else \[ G = G \land \neg A \]
DPLL($T$)

\[ \text{DPLL}_T(F) \]
\[
G = \mathcal{B}(F) \\
\text{while (true) do} \\
\quad A, \text{out} = \text{SAT-SOLVER}(G) \\
\quad \text{if (out = UNSAT) then return UNSAT} \\
\quad \text{else} \\
\quad \quad \text{out} = \text{T-SOLVER} (\mathcal{B}^{-1}(A)) \\
\quad \quad \text{if (out = SAT) then return SAT} \\
\quad \quad \text{else } G = G \land \neg A
\]
DPLL($T$)

DPLL$_T$(F)

$G = \mathcal{B}(F)$

while (true) do

$A, \text{out} = \text{SAT-SOLVER}(G)$

if (out = UNSAT) then return UNSAT
else

out = $T$-SOLVER($\mathcal{B}^{-1}(A)$)

if (out = SAT) then return SAT
else $G = G \land \lnot A$

Boolean abstraction

conjunction of propositional literals

conjunction of atomic $T$-formulas
\textbf{DPLL(T)}

\[ \text{DPLL}_T(F) \]
\[ G = \mathcal{B}(F) \]
\[ \text{while } (\text{true}) \text{ do} \]
\[ A, \text{out} = \text{SAT-SOLVER}(G) \]
\[ \text{if } (\text{out} = \text{UNSAT}) \text{ then return } \text{UNSAT} \]
\[ \text{else} \]
\[ \text{out} = \text{T-SOLVER} (\mathcal{B}^{-1}(A)) \]
\[ \text{if } (\text{out} = \text{SAT}) \text{ then return } \text{SAT} \]
\[ \text{else } G = G \land \neg A \]

- Boolean abstraction
- Conjunction of propositional literals
- Conjunction of atomic $T$-formulas
- Theory conflict clause
DPLL(T)

\[
DPLL_T(F) \quad G = B(F) \\
\text{while (true) do} \\
A, \text{out} = \text{SAT-SOLVER}(G) \\
\text{if (out = UNSAT) then return UNSAT} \\
\text{else} \\
\text{out} = T\text{-SOLVER } (B^{-1}(A)) \\
\text{if (out = SAT) then return SAT} \\n\text{else } G = G \land \neg A
\]
DPLL($T$): improvement

\texttt{DPLL}_T(F)

\[ G = \texttt{B}(F) \]

\textbf{while (true) do}

\[ A, \texttt{out} = \texttt{SAT-SOLVER}(G) \]

\textbf{if (out = UNSAT) then return UNSAT}

\textbf{else}

\[ \texttt{out} = \texttt{T-SOLVER}(\texttt{B}^{-1}(A)) \]

\textbf{if (out = SAT) then return SAT}

\textbf{else}

\[ G = G \land \neg \texttt{B}(-\texttt{MINIMAL UNSAT CORE}(\texttt{B}^{-1}(A))) \]
DPLL($T$): improvement

DPLL$_T$(F)

\[ G = B(F) \]

while (true) do

\[ A, \text{out} = \text{SAT-SOLVER}(G) \]

if (out = UNSAT) then return UNSAT
else

\[ \text{out} = T\text{-SOLVER } (B^{-1}(A)) \]

if (out = SAT) then return SAT
else \[ G = G \land \neg B(\text{MINIMAL UNSAT CORE}(B^{-1}(A))) \]

An unsatisfiable core $C$ of $A$ contains a subset of atoms in $A$ such that $B^{-1}(C)$ is still unsatisfiable.

Minimal unsatisfiable core $C^*$ has the property that if you drop any single atom of $C^*$, result is satisfiable.
DPLL\((T)\): improvement

\[ \text{DPLL}_T(F) \]

\[ G = \mathcal{B}(F) \]

\text{while (true) do}

\[ A, \text{out} = \text{SAT-Solver}(G) \]

\text{if (out = UNSAT) then return UNSAT}

\text{else}

\[ \text{out} = \text{T-Solver} (\mathcal{B}^{-1}(A)) \]

\text{if (out = SAT) then return SAT}

\text{else} \quad G = G \land \neg \mathcal{B}(\text{MINIMAL UNSAT CORE}(\mathcal{B}^{-1}(A)))

An unsatisfiable core \( C \) of \( A \) contains a subset of atoms in \( A \) such that \( \mathcal{B}^{-1}(C) \) is still unsatisfiable.

Minimal unsatisfiable core \( C^* \) has the property that if you drop any single atom of \( C^* \), result is satisfiable.

Waits for \textit{full} assignment to the Boolean abstraction to generate conflict clause.
DPLL($T$): improvement

\[ \text{DPLL}_T(F) \]

\[ G = B(F) \]

\text{while (true) do}

\[ A, \text{out} = \text{SAT-SOLVER}(G) \]

\text{if (out = UNSAT) then return UNSAT}

\text{else}

\[ \text{out} = T-\text{SOLVER} (B^{-1}(A)) \]

\text{if (out = SAT) then return SAT}

\text{else} \ G = G \land \neg B(\text{MINIMAL UNSAT CORE}(B^{-1}(A)))

An unsatisfiable core $C$ of $A$ contains a subset of atoms in $A$ such that $B^{-1}(C)$ is still unsatisfiable.

Minimal unsatisfiable core $C^*$ has the property that if you drop any single atom of $C^*$, result is satisfiable.

Waits for full assignment to the Boolean abstraction to generate conflict clause.

Solution: Integrate theory solver into DPLL. Don’t use SAT solver as “blackbox”.

An unsatisfiable core $C$ of $A$ contains a subset of atoms in $A$ such that $B^{-1}(C)$ is still unsatisfiable.
$x = g(y)$

Core solver: Equality and UF

<table>
<thead>
<tr>
<th>Equation</th>
<th>Theory Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2x + y \leq 5$</td>
<td>Linear Real Arithmetic</td>
</tr>
<tr>
<td>$2i + j \leq 5$</td>
<td>Linear Integer Arithmetic</td>
</tr>
<tr>
<td>$(b \gg 2) = c$</td>
<td>Fixed-Width Bitvectors</td>
</tr>
<tr>
<td>$a[i] = x$</td>
<td>Arrays</td>
</tr>
</tbody>
</table>
Theory of equality $T_=$

**Signature**

$$\Sigma_:= \{=, a, b, c, \ldots, f, g, h, \ldots, p, q, r\}$$

**Axioms**

1. $\forall x. \ x = x$ (reflexivity)
2. $\forall x, y. \ (x = y) \rightarrow y = x$ (symmetry)
3. $\forall x, y, z. \ (x = y \land y = z) \rightarrow x = z$ (transitivity)
4. $\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \ (\land_i x_i = y_i) \rightarrow (f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$ (fn. congruence)
5. $\forall x_1, \ldots, x_n, y_1, \ldots y_n. \ (\land_i x_i = y_i) \rightarrow ((p(x_1, \ldots, x_n) \leftrightarrow p(y_1, \ldots, y_n))$ (pr. congruence)
Theory of equality $T_\equiv$

Signature

$$\Sigma_\equiv := \{=, a, b, c, \ldots, f, g, h, \ldots, p, q, r\}$$

Axioms

1. $\forall x. \ x = x$ (reflexivity)
2. $\forall x, y. \ (x = y) \rightarrow y = x$ (symmetry)
3. $\forall x, y, z. \ (x = y \land y = z) \rightarrow x = z$ (transitivity)
4. $\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \ (\land_i x_i = y_i) \rightarrow (f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$ (fn. congruence)
5. $\forall x_1, \ldots, x_n, y_1, \ldots y_n. \ (\land_i x_i = y_i) \rightarrow ((p(x_1, \ldots, x_n) \leftrightarrow p(y_1, \ldots, y_n))$ (pr. congruence)

Eliminate predicates to get equisatisfiable formula with only functions

Introduce fresh constant $\bullet$

For each $p$:
1. introduce a fresh function constant $f_p$
2. $p(x_1, \ldots, x_n) \rightarrow f_p(x_1, \ldots, x_n) = \bullet$
Theory of equality & uninterpreted functions $T_=\text{TEUF}$

Signature

$\Sigma_= \{=, a, b, c, \ldots, f, g, h\}$

Axioms

1. $\forall x. \ x = x$ (reflexivity)
2. $\forall x, y. \ (x = y) \rightarrow y = x$ (symmetry)
3. $\forall x, y, z. \ (x = y \land y = z) \rightarrow x = z$ (transitivity)
4. $\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \ (\land_i x_i = y_i) \rightarrow (f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$ (fn. congruence)

$T_=\text{models}$

All first-order structures $\langle U, I \rangle$ that satisfy the axioms of $T_=$
Is a conjunction of $T_\equiv$ literals satisfiable?

\[ f\left(f\left(f(a)\right)\right) = a \land f\left(f\left(f\left(f\left(f(a)\right)\right)\right)\right) = a \land f(a) \neq a \]
Is a conjunction of $T$ literals satisfiable?

$f(f(f(a))) = a \land f(f(f(f(a)))) = a \land f(a) \neq a$

i.e, $f^3(a) = a \land f^5(a) = a \land f(a) \neq a$
Is a conjunction of $T_\neq$ literals satisfiable?

\[
f(f(f(a))) = a \land f\left(f\left(f\left(f(f(a))\right)\right)\right) = a \land f(a) \neq a
\]

i.e, $f^3(a) = a \land f^5(a) = a \land f(a) \neq a$

Decision procedure: Congruence closure algorithm
Congruence closure algorithm: basic sketch

Place each subterm of $F$ into its own congruence class.

For each positive literal $t_1 = t_2$ in $F$:
- Merge the classes for $t_1$ and $t_2$
- Propagate the resulting congruences

If $F$ has a negative literal $t_1 \neq t_2$ with $t_1$ and $t_2$ in the same congruence class, output UNSAT

Otherwise, output SAT
Congruence closure algorithm: basic sketch

Place each subterm of $F$ into its own congruence class.

For each positive literal $t_1 = t_2$ in $F$:
- Merge the classes for $t_1$ and $t_2$
- Propagate the resulting congruences

If $F$ has a negative literal $t_1 \neq t_2$ with $t_1$ and $t_2$ in the same congruence class, output UNSAT

Otherwise, output SAT

Computing the “congruence closure” of $=$ over the subterm set
Congruence closure algorithm: data structure

- Represent subterm set as a DAG: each node corresponds to a subterm and edges point from function symbol to arguments
- Each node stores its unique id, name of function or variable, and list of arguments

\[ f(a, b) = a \land f(f(a, b), b) \neq a \]
Congruence closure algorithm: data structure

- Represent subterm set as a DAG: each node corresponds to a subterm and edges point from function symbol to arguments.
- Each node stores its unique id, name of function or variable, and list of arguments.
- Each node $n$ has a `find` pointer field that leads to the representative of its congruence class (or to itself if it is the representative).

\[
f(a, b) = a \land f(f(a, b), b) \neq a
\]
Congruence closure algorithm: data structure

- Represent subterm set as a DAG: each node corresponds to a subterm and edges point from function symbol to arguments
- Each node stores its unique id, name of function or variable, and list of arguments
- Each node $n$ has a `find` pointer field that leads to the representative of its congruence class (or to itself if it is the representative)

Each congruence class has one representative. When merging two classes, only need to update the representative.
Congruence closure algorithm: data structure

- Represent subterm set as a DAG: each node corresponds to a subterm and edges point from function symbol to arguments.
- Each node stores its unique id, name of function or variable, and list of arguments.
- Each node $n$ has a `find` pointer field that leads to the representative of its congruence class (or to itself if it is the representative).
- Each representative has a `ccpar` field that stores the set of parents for all subterms in its congruence class.

If $x_1 = y_1, \ldots, x_k = y_k$, need to merge congruence classes of their parents $f(\vec{x})$ and $f(\vec{y})$. 

```
1: f
   ↭
2: f
   ↭
3: a
   ↭
   \{1,2\}
4: b
   ↭
   \{1,2\}
```
Congruence closure algorithm

\textbf{DECIDE}(F)

- construct the DAG for \( F \)'s subterms
  
  \textbf{for} \( s_i = t_i \in F \)
  
  \textbf{MERGE}(s_i, t_i)
  
  \textbf{for} \( s_i \neq t_i \in F \)
  
  \textbf{if} FIND\( (s_i) = \) FIND\( (t_i) \)
  
  \textbf{then} return UNSAT

return SAT
Congruence closure algorithm

\textbf{Decide}(F)

\begin{itemize}
  \item construct the DAG for \( F \)'s subterms
  \item for \( s_i = t_i \in F \) \quad \textbf{Merge}(s_i, t_i)
  \item for \( s_i \neq t_i \in F \)
    \begin{itemize}
      \item if \( \text{Find}(s_i) = \text{Find}(t_i) \)
        \begin{itemize}
          \item then return UNSAT
        \end{itemize}
    \end{itemize}
\end{itemize}

return SAT

Place each subterm of \( F \) into its own congruence class.

For each positive literal \( t_1 = t_2 \) in \( F \):
  \begin{itemize}
  \item Merge the classes for \( t_1 \) and \( t_2 \)
  \item Propagate the resulting congruences
  \end{itemize}

If \( F \) has a negative literal \( t_1 \neq t_2 \) with \( t_1 \) and \( t_2 \) in the same congruence class, output UNSAT.

Otherwise, output SAT.
Congruence closure algorithm

\textbf{Decide}(F)

construct the DAG for \(F\)'s subterms

\begin{itemize}
  \item for \( s_i = t_i \in F \)
  \hspace{1em} \text{\textbf{Merge}}(s_i, t_i)
  \item for \( s_i \neq t_i \in F \)
  \hspace{1em} if \text{\textbf{Find}}(s_i) = \text{\textbf{Find}}(t_i)
  \hspace{1em} then return \text{UNSAT}
\end{itemize}

return \text{SAT}

Place each subterm of \(F\) into its own congruence class.

For each positive literal \( t_1 = t_2 \) in \(F\):
\begin{itemize}
  \item Merge the classes for \( t_1 \) and \( t_2 \)
  \item Propagate the resulting congruences
\end{itemize}

If \(F\) has a negative literal \( t_1 \neq t_2 \) with \( t_1 \) and \( t_2 \) in the same congruence class, output \text{UNSAT}

Otherwise, output \text{SAT}
**Congruence closure algorithm**

\[\text{\textbf{DECIDE}}(F)\]

- construct the DAG for \(F\)'s subterms
  
  for \( s_i = t_i \in F \)
  
  \[\text{\textbf{MERGE}}(s_i, t_i)\]

  for \( s_i \neq t_i \in F \)
  
  if \( \text{FIND}(s_i) = \text{FIND}(t_i) \)
  
  then return \text{UNSAT}

  return \text{SAT}

Place each subterm of \(F\) into its own congruence class.

For each positive literal \( t_1 = t_2 \) in \(F\):  
- Merge the classes for \( t_1 \) and \( t_2 \)
- Propagate the resulting congruences

If \(F\) has a negative literal \( t_1 \neq t_2 \) with \( t_1 \) and \( t_2 \) in the same congruence class, output \text{UNSAT}

Otherwise, output \text{SAT}
**Congruence closure algorithm**

**DECIDE**(F)

1. construct the DAG for F’s subterms
   
2. for \( s_i = t_i \in F \)
   
   **MERGE**(s_i, t_i)
   
3. for \( s_i \neq t_i \in F \)
   
   if **FIND**(s_i) = **FIND**(t_i)
   
   then return UNSAT

   return SAT

Place each subterm of F into its own congruence class.

For each positive literal \( t_1 = t_2 \) in F:

- Merge the classes for \( t_1 \) and \( t_2 \)
- Propagate the resulting congruences

If F has a negative literal \( t_1 \neq t_2 \) with \( t_1 \) and \( t_2 \) in the same congruence class, output UNSAT

Otherwise, output SAT
**Congruence closure algorithm**

**\text{Decide}(F)**

- Construct the DAG for $F$'s subterms.
  
  **\text{for}** $s_i = t_i \in F$
  
  **\text{Merge}(s_i, t_i)**
  
  **\text{for}** $s_i \neq t_i \in F$
  
  **\text{if}** $\text{Find}(s_i) = \text{Find}(t_i)$
  
  **then return** \text{UNSAT}
  
  **return** \text{SAT}

- Place each subterm of $F$ into its own congruence class.

  For each positive literal $t_1 = t_2$ in $F$:
  
  - Merge the classes for $t_1$ and $t_2$
  
  - Propagate the resulting congruences

- If $F$ has a negative literal $t_1 \neq t_2$ with $t_1$ and $t_2$ in the same congruence class, output \text{UNSAT}

- Otherwise, output \text{SAT}
**Congruence closure algorithm: union-find**

**FIND** returns the representative of a node’s congruence class by following **find** pointers until it finds a self-loop.

\[
\begin{align*}
\text{1:} & \quad f(a, b) = a \land f(f(a, b), b) \neq a \\
\text{2:} & \\
\text{3:} & \quad 3:a \\
\text{4:} & \quad 4:b \\
\end{align*}
\]
**Congruence closure algorithm: union-find**

\[ f(a, b) = a \quad \land \quad f(f(a, b), b) \neq a \]

**FIND** returns the representative of a node’s congruence class by following **find** pointers until it finds a self-loop.

![Diagram of FIND(2)? and congruence classes]
**Congruence closure algorithm: union-find**

**FIND** returns the representative of a node’s congruence class by following **find** pointers until it finds a self-loop.

**UNION** combines congruence classes for nodes $i_1$ and $i_2$:

$n_1, n_2 = \text{FIND}(i_1), \text{FIND}(i_2)$

$n_1.\text{find} = n_2$

$n_2.\text{ccp} = n_1.\text{ccp} \cup n_2.\text{ccp}$

$n_1.\text{ccp} = \emptyset$

$f(a, b) = a \land f(f(a, b), b) \neq a$
**Congruence closure algorithm: union-find**

**FIND** returns the representative of a node’s congruence class by following **find** pointers until it finds a self-loop.

**UNION** combines congruence classes for nodes $i_1$ and $i_2$:

$$n_1, n_2 = \text{FIND}(i_1), \text{FIND}(i_2)$$
$$n_1 \cdot \text{find} = n_2$$
$$n_2 \cdot \text{ccp} = n_1 \cdot \text{ccp} \cup n_2 \cdot \text{ccp}$$
$$n_1 \cdot \text{ccp} = \emptyset$$

$f(a, b) = a \land f(f(a, b), b) \neq a$
**Congruence closure algorithm: congruent**

**CONGRUENT** take as input two nodes and return true iff their:
- functions are the same
- corresponding arguments are in the same congruence class

\[
f(a, b) = a \land f(f(a, b), b) \neq a
\]
Congruence closure algorithm: congruent

\text{CONGRUENT} \text{ take as input two nodes and return true iff their:}

\begin{itemize}
  \item functions are the same
  \item corresponding arguments are in the same congruence class
\end{itemize}

\hspace{1cm}

\hspace{1cm}

\begin{align*}
  f(a, b) &= a \land f(f(a, b), b) \neq a
\end{align*}

\hspace{1cm}

\hspace{1cm}

\hspace{1cm}

\hspace{1cm}
Congruence closure algorithm: merge

\[
\text{MERGE}(i_1, i_2) \\
\quad n_1, n_2 = \text{FIND}(i_1), \text{FIND}(i_2) \\
\quad \text{if } n_1 = n_2 \text{ then return } \\
\quad p_1, p_2 = n_1\cdot\text{ccp}, n_2\cdot\text{ccp} \\
\quad \text{UNION}(n_1, n_2) \\
\quad \text{for each } t_1, t_2 \in p_1 \times p_2 \\
\quad \quad \text{if } \text{FIND}(t_1) \neq \text{FIND}(t_2) \land \text{CONGRUENT}(t_1, t_2) \\
\quad \quad \text{then } \text{MERGE}(t_1, t_2)
\]

\[f(a, b) = a \land f(f(a, b), b) \neq a\]
Congruence closure algorithm: merge

\textbf{MERGE}(i_1, i_2)
\begin{align*}
n_1, n_2 &= \text{FIND}(i_1), \text{FIND}(i_2) \\
\text{if } n_1 = n_2 \text{ then return} \\
p_1, p_2 &= n_1 \cdot \text{ccp}, n_2 \cdot \text{ccp} \\
\text{UNION}(n_1, n_2) \\
\text{for each } t_1, t_2 \in p_1 \times p_2 \\
\quad \text{if } \text{FIND}(t_1) \neq \text{FIND}(t_2) \land \text{CONGRUENT}(t_1, t_2) \\
\quad \text{then MERGE}(t_1, t_2)
\end{align*}

\hspace{2cm}

\begin{align*}
f(a, b) &= a \\
\land f(f(a, b), b) &\neq a
\end{align*}
Congruence closure algorithm: merge

\(\text{MERGE}(i_1, i_2)\)

\(n_1, n_2 = \text{FIND}(i_1), \text{FIND}(i_2)\)

if \(n_1 = n_2\) then return

\(p_1, p_2 = n_1 \cdot \text{ccp}, n_2 \cdot \text{ccp}\)

\(\text{UNION}(n_1, n_2)\)

for each \(t_1, t_2 \in p_1 \times p_2\)

if \(\text{FIND}(t_1) \neq \text{FIND}(t_2) \land \text{CONGRUENT}(t_1, t_2)\)
then \(\text{MERGE}(t_1, t_2)\)

\(f(a, b) = a \land f(f(a, b), b) \neq a\)
Congruence closure algorithm: merge

\[
\text{M} \text{ERGE}(i_1, i_2)
\]

\[
\begin{align*}
n_1, n_2 &= \text{FIND}(i_1), \text{FIND}(i_2) \\
\text{if } n_1 &= n_2 \text{ then return} \\
p_1, p_2 &= n_1 \cdot \text{ccp}, n_2 \cdot \text{ccp} \\
\text{UNION}(n_1, n_2) \\
\text{for each } t_1, t_2 &\in p_1 \times p_2 \\
\text{if } \text{FIND}(t_1) &\neq \text{FIND}(t_2) \land \text{CONGRUENT}(t_1, t_2) \\
\text{then } \text{M} \text{ERGE}(t_1, t_2)
\end{align*}
\]

\[
f(a, b) = a \land f(f(a, b), b) \neq a
\]
Congruence closure algorithm: merge

\[
\text{MERGE}(i_1, i_2)
\]

\[
n_1, n_2 = \text{FIND}(i_1), \text{FIND}(i_2)
\]

\[
\text{if } n_1 = n_2 \text{ then return}
\]

\[
p_1, p_2 = n_1 \cdot \text{ccp}, n_2 \cdot \text{ccp}
\]

\[
\text{UNION}(n_1, n_2)
\]

\[
\text{for each } t_1, t_2 \in p_1 \times p_2
\]

\[
\text{if } \text{FIND}(t_1) \neq \text{FIND}(t_2) \land \text{CONGRUENT}(t_1, t_2)
\]

\[
\text{then } \text{MERGE}(t_1, t_2)
\]

\[
f(a, b) = a \land f(f(a, b), b) \neq a
\]
Congruence closure algorithm

\textbf{DECIDE}(F)

construct the DAG for \( F \)'s subterms

\begin{itemize}
  \item for \( s_i = t_i \in F \)
  \begin{itemize}
    \item \textbf{MERGE}(s_i, t_i)
  \end{itemize}
  \item for \( s_i \neq t_i \in F \)
  \begin{itemize}
    \item if \textbf{FIND}(s_i) = \textbf{FIND}(t_i)
    \begin{itemize}
      \item then return \text{UNSAT}
    \end{itemize}
  \end{itemize}
\end{itemize}

return \text{SAT}

\[ f(a, b) = a \land f(f(a, b), b) \neq a \]
**Congruence closure algorithm**

\[ f(a, b) = a \land f(f(a, b), b) \neq a \]

\[
\text{DECEDE}(F)
\]

construct the DAG for \( F \)'s subterms

\[
\text{for } s_i = t_i \in F
\]

\[
\text{MERGE}(s_i, t_i)
\]

\[
\text{for } s_i \neq t_i \in F
\]

\[
\text{if } \text{FIND}(s_i) = \text{FIND}(t_i)
\]

\[
\text{then return } \text{UNSAT}
\]

\[
\text{return } \text{SAT}
\]
Congruence closure algorithm

**DECIDE**($F$)

- construct the DAG for $F$’s subterms

  for $s_i = t_i \in F$
  - **MERGE**($s_i, t_i$)

  for $s_i \neq t_i \in F$
  - if **FIND**($s_i$) = **FIND**($t_i$)
  - then return UNSAT

- return SAT

\[
f(a, b) = a \land f(f(a, b), b) \neq a
\]
Congruence closure algorithm

\[ f(a, b) = a \land f(f(a, b), b) \neq a \]

**DECIDE**\( (F) \)

- construct the DAG for \( F \)'s subterms
- for \( s_i = t_i \in F \)
  - **MERGE**\( (s_i, t_i) \)
- for \( s_i \neq t_i \in F \)
  - if **FIND**\( (s_i) = **FIND** (t_i) \)
    - then return **UNSAT**
  - return **SAT**
**Congruence closure algorithm**

**DECEIDE**($F$)

- construct the DAG for $F$’s subterms
- for $s_i = t_i \in F$
  - MERGE($s_i$, $t_i$)
- for $s_i \neq t_i \in F$
  - if FIND($s_i$) = FIND($t_i$)
    - then return UNSAT
  - return SAT

\[ f(a, b) = a \land f(f(a, b), b) \neq a \]
A binary relation $R$ over a set $S$ is an **equivalence relation** if it is

1. reflexive: $\forall s \in S. R(s, s)
2. symmetric: $\forall s_1, s_2 \in S. R(s_1, s_2) \rightarrow R(s_2, s_1)
3. transitive: $\forall s_1, s_2, s_3 \in S. R(s_1, s_2) \land R(s_2, s_3) \rightarrow R(s_1, s_3)

The **equivalence class** of element $s \in S$ under $R$: $[s]_R \triangleq \{ s' \in S : R(s, s') \}$

A equivalence relation $R$ over a set $S$ is a **congruence relation** if for every $n$-ary function $f$ :

$$\forall \vec{s}, \vec{t}. \bigwedge_{i=1}^{n} R(s_i, t_i) \rightarrow R(f(\vec{s}), f(\vec{t}))$$

The **congruence class** of element $s \in S$ under $R$ is its equivalence class
A binary relation $R_1$ is a refinement of another binary relation $R_2$, written $R_1 \prec R_2$, if
\[
\forall s_1, s_2 \in S. \ R_1(s_1, s_2) \rightarrow R_2(s_1, s_2)
\]

The equivalence closure $R^E$ of a binary relation $R$ over $S$ is the equivalence relation such that:
1. $R$ refines $R^E$, i.e. $R < R^E$;
2. for all other equivalence relations $R'$ with $R < R'$, either $R' = R^E$ or $R^E < R'$

The congruence closure $R^C$ of a binary relation $R$ over $S$ is the congruence relation such that:
1. $R$ refines $R^C$, i.e. $R < R^C$;
2. for all other congruence relations $R'$ s.t. $R < R'$, either $R' = R^C$ or $R^C < R'$
A binary relation $R_1$ is a refinement of another binary relation $R_2$, written $R_1 \prec R_2$, if
\[ \forall s_1, s_2 \in S. \ R_1(s_1, s_2) \rightarrow R_2(s_1, s_2) \]

The equivalence closure $R^E$ of a binary relation $R$ over $S$ is the equivalence relation such that:
1. $R$ refines $R^E$, i.e. $R \prec R^E$;
2. for all other equivalence relations $R'$ with $R \prec R'$, either $R' = R^E$ or $R^E \prec R'$

The congruence closure $R^C$ of a binary relation $R$ over $S$ is the congruence relation such that:
1. $R$ refines $R^C$, i.e. $R \prec R^C$;
2. for all other congruence relations $R'$ s.t. $R \prec R'$, either $R' = R^C$ or $R^C \prec R'$
Satisfiability using congruence relations

Let $F$ be a $\Sigma_\equiv$ formula as follows:
$$s_1 = t_1 \land \ldots \land s_m = t_m \land s_{m+1} \neq t_{m+1} \land \ldots \land s_n \neq t_n$$

$F$ is satisfiable iff there exists a congruence relation $\sim$ over the subterm set $S_F$ of $F$ such that:

1. For each $i$ in $[1, m]$, $s_i \sim t_i$
2. For each $i$ in $[m + 1, n]$, $s_i \nsim t_i$
Satisfiability using congruence relations

Let $F$ be a $\Sigma_\equiv$ formula as follows:

$$s_1 = t_1 \land \ldots \land s_m = t_m \land s_{m+1} \neq t_{m+1} \land \ldots \land s_n \neq t_n$$

$F$ is satisfiable iff there exists a congruence relation $\sim$ over the subterm set $S_F$ of $F$ such that:

1. For each $i$ in $[1, m]$, $s_i \sim t_i$
2. For each $i$ in $[m + 1, n]$, $s_i \not\sim t_i$

The congruence closure algorithm computes such a congruence relation $\sim$, or, proves that no such relation exists.
Summary

Today

- SMT solving
- DPLL(T) : combine DPLL algorithm for SAT solving with theory solvers
- A core theory solver: congruence closure algorithm for $T_=$

Next

- Temporal logic
\[
R_1 = R_1 \prec \left( R_1 \cup R_2 \right)
\]
\[
R_2 = R_2 \prec \left( R_1 \cup R_2 \right)
\]