SMT Solving
A Core Theory Solver

CS560: Reasoning About Programs

Roopsha Samanta

PURDUE UNIVERSITY

Partly based on slides by Isil Dillig and Emina Torlak

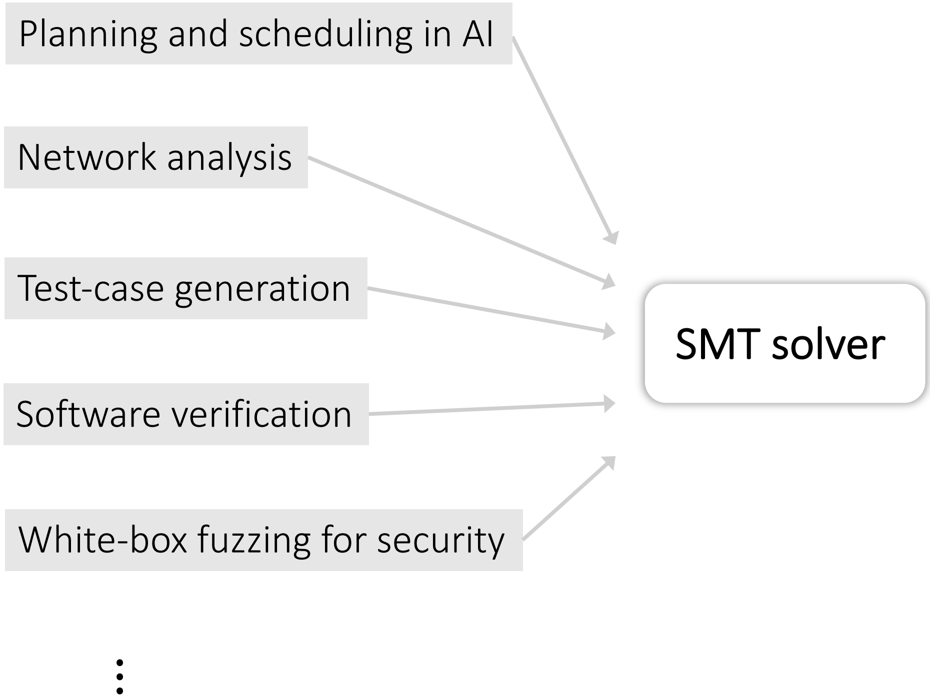
Roadmap

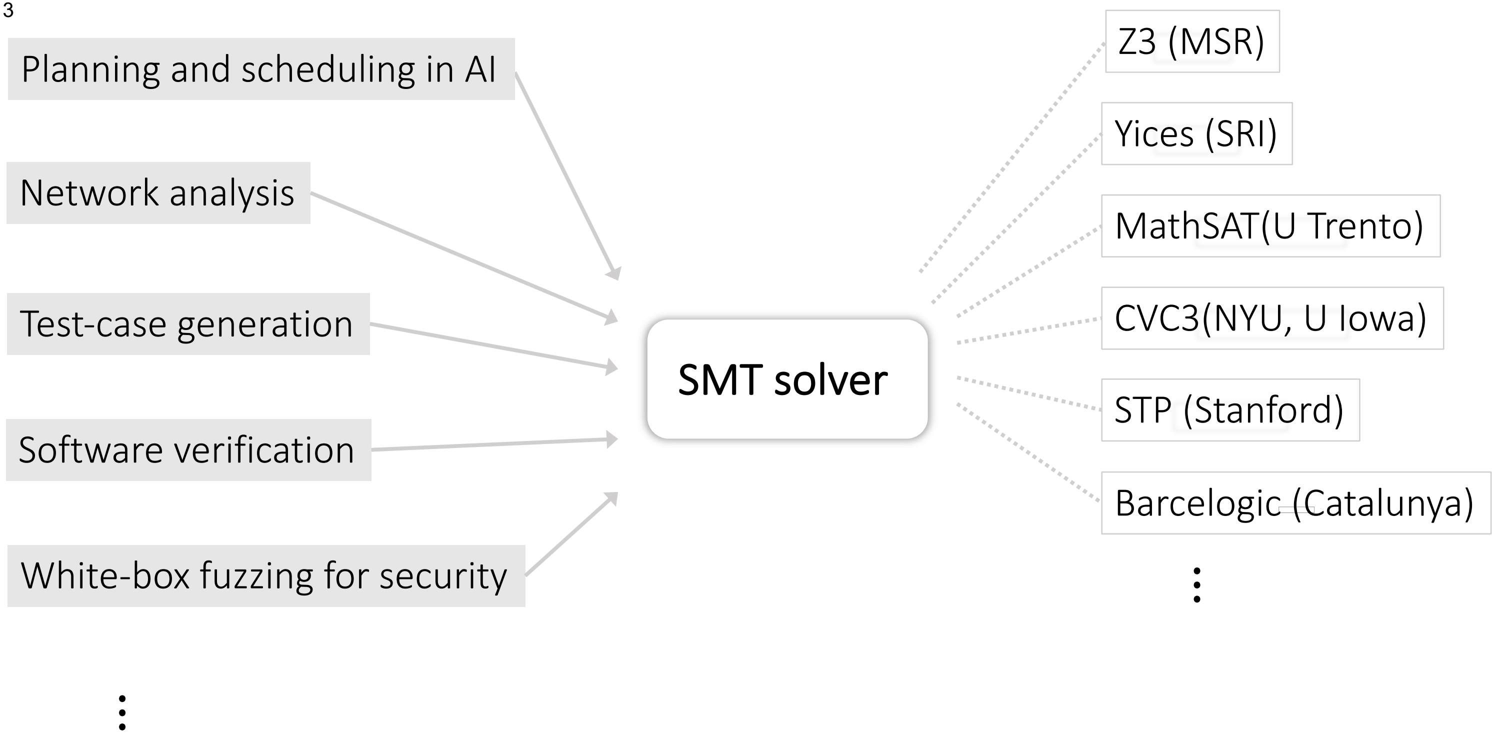
Previously

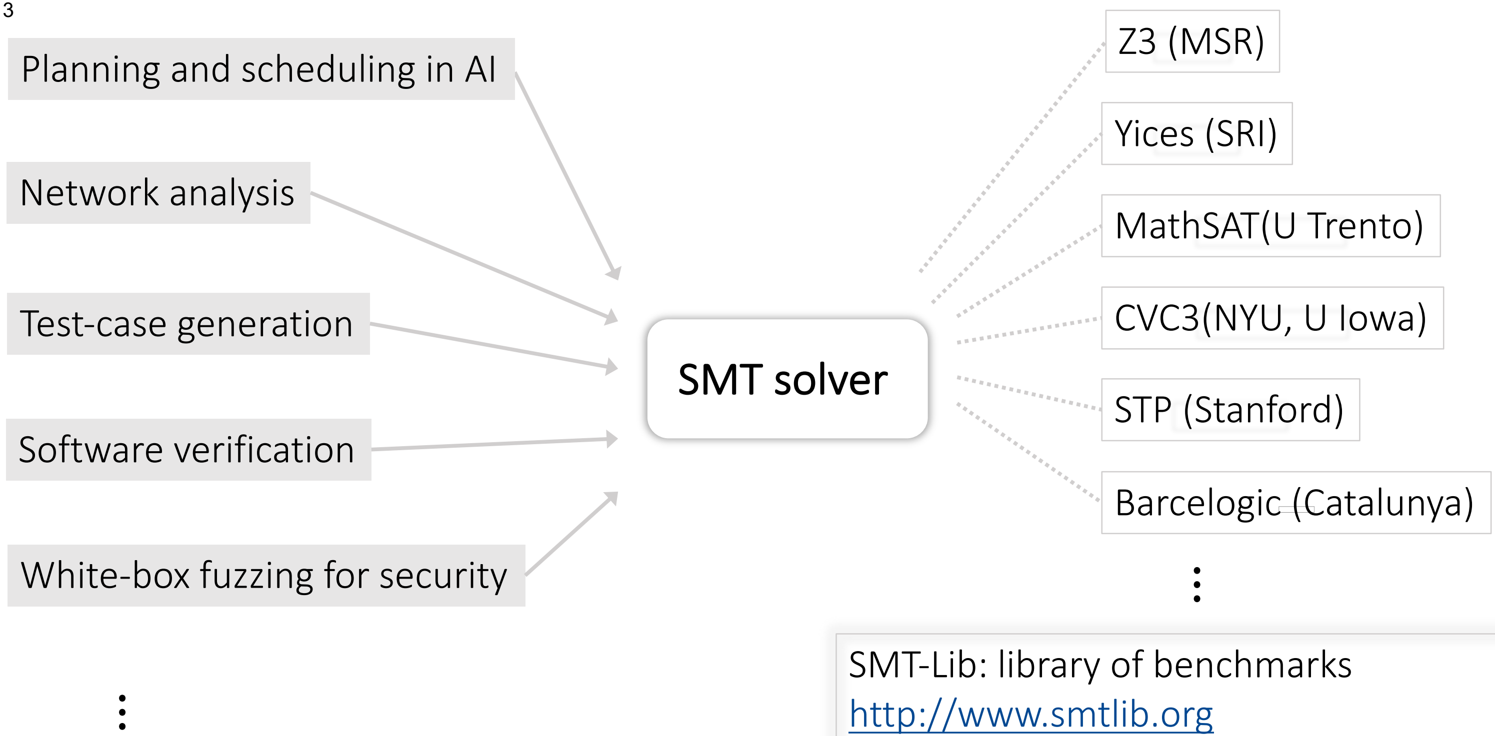
- Propositional logic and SAT solving
- First-order logic and first-order theories

Today

- SMT solving
- DPLL(T) : Combine DPLL algorithm for SAT solving with theory solvers
- A core theory solver: congruence closure algorithm for $\boldsymbol{T}_{=}$

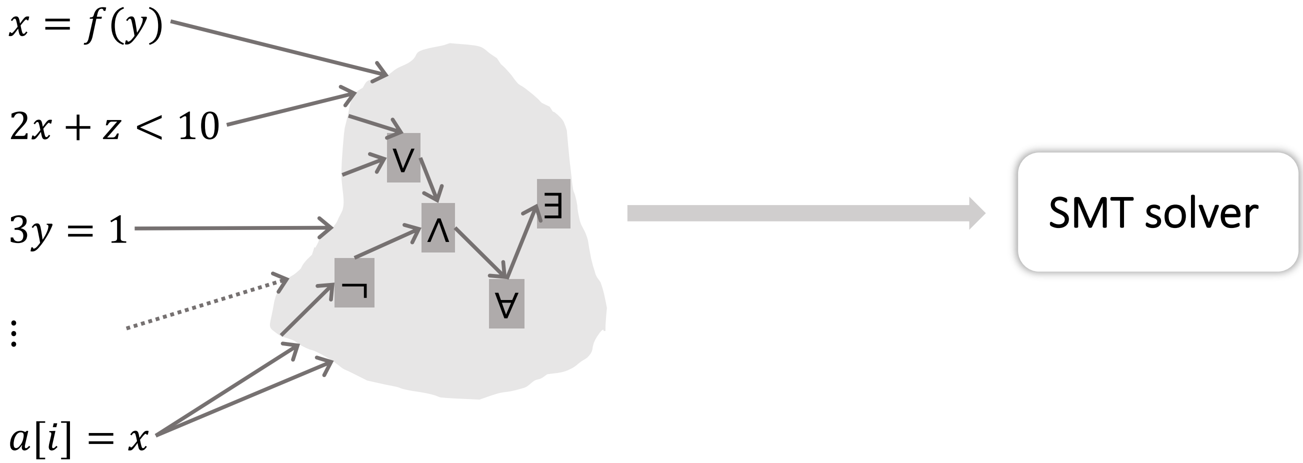

Planning and scheduling in AI
Network analysis
Test-case generation
Software verification
White-box fuzzing for security
⋮
SMT solver


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)
⋮

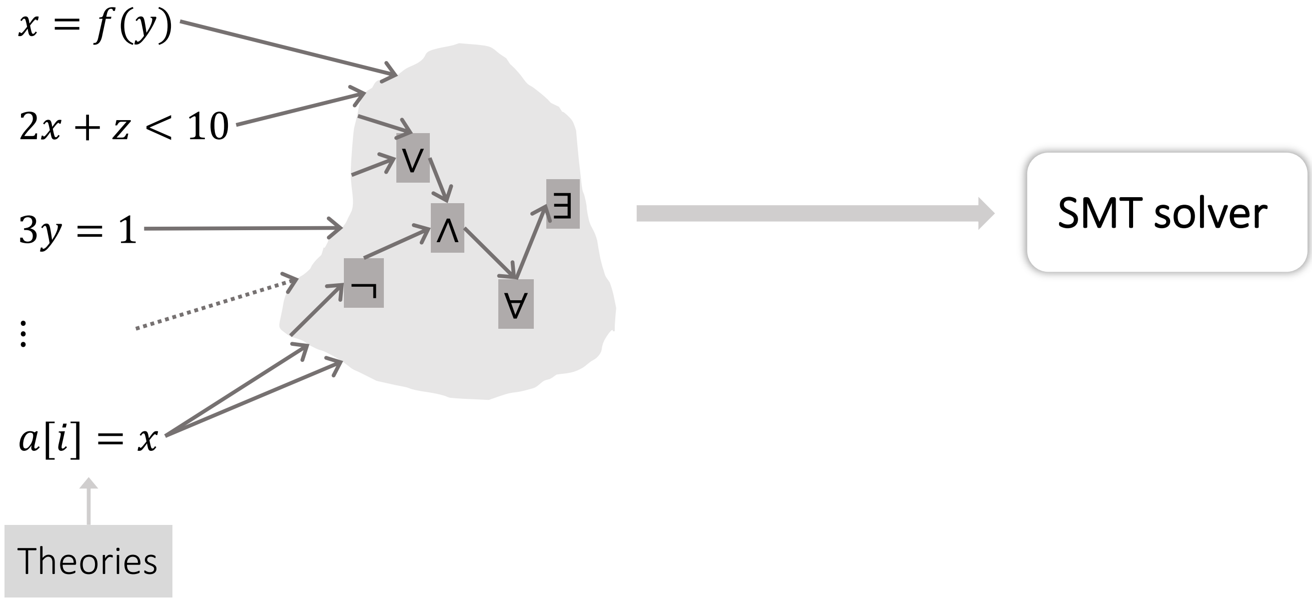


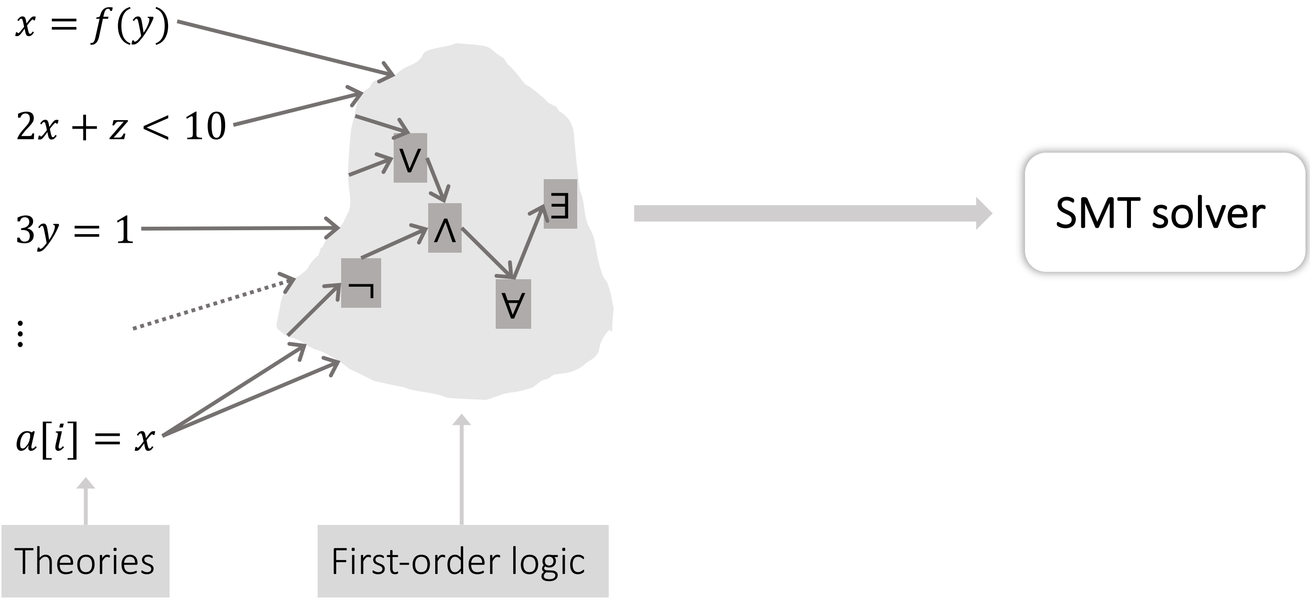
SMT-Lib: library of benchmarks
http://www.smtlib.org

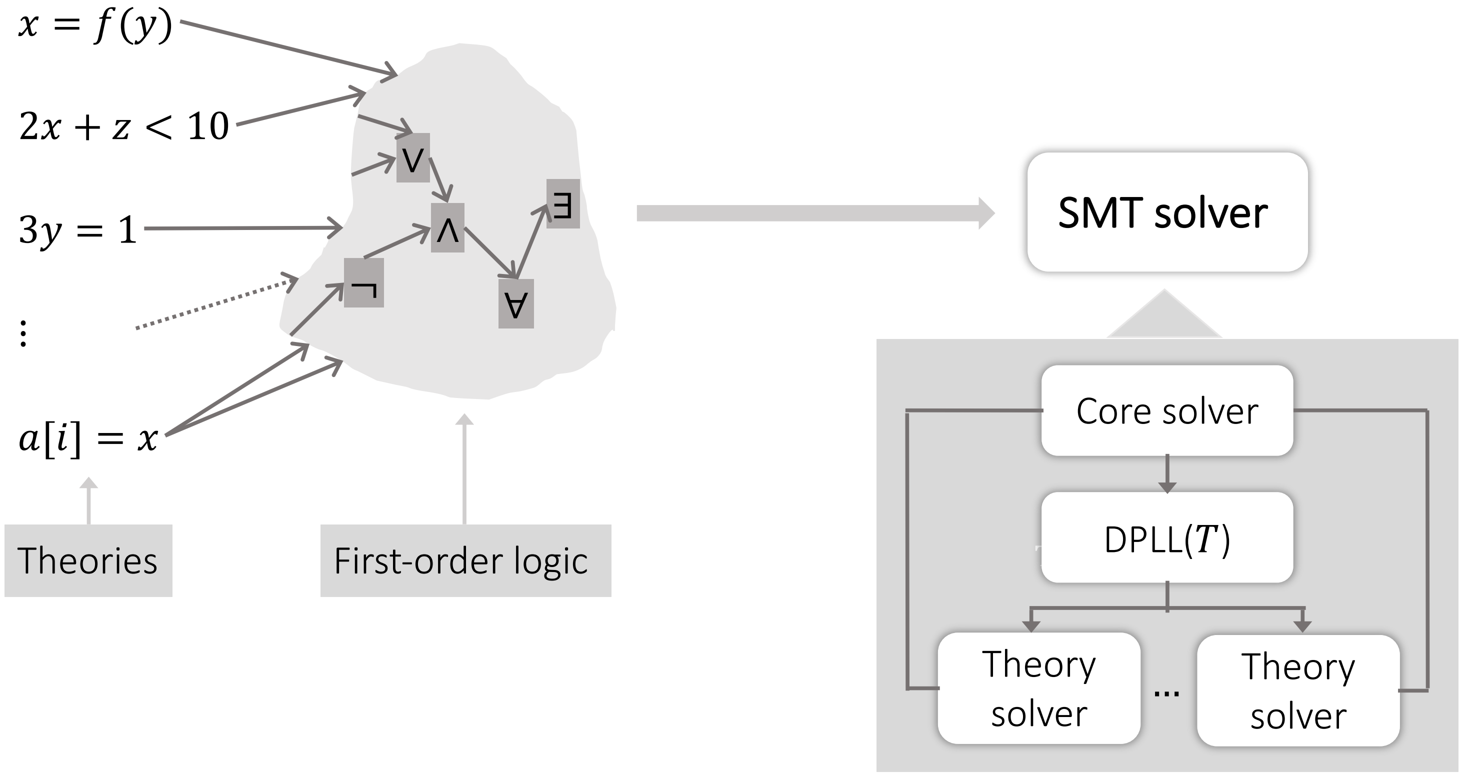
SMT-Comp: annual SMT-Solver competition
http://www.smtcomp.org

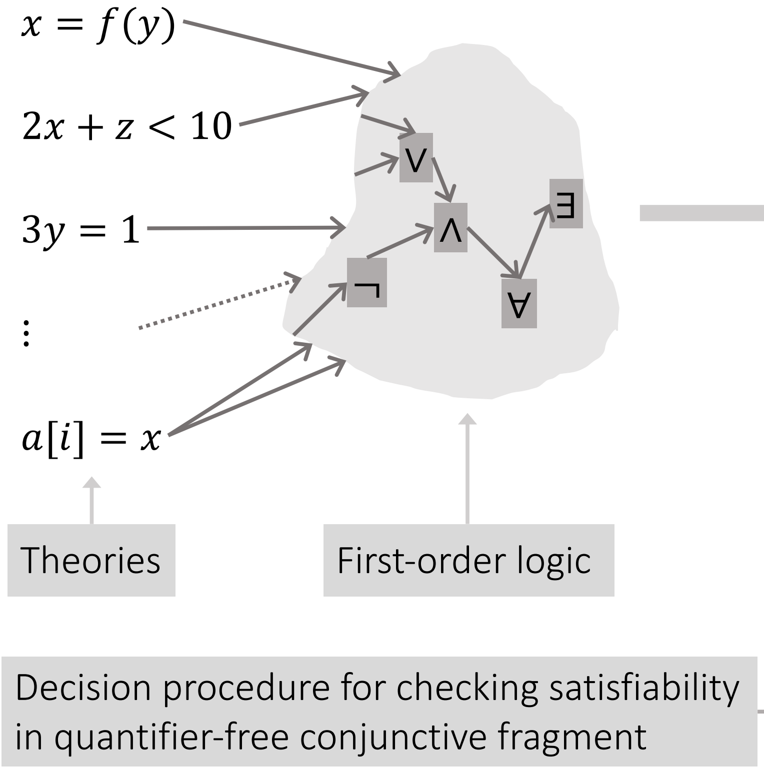


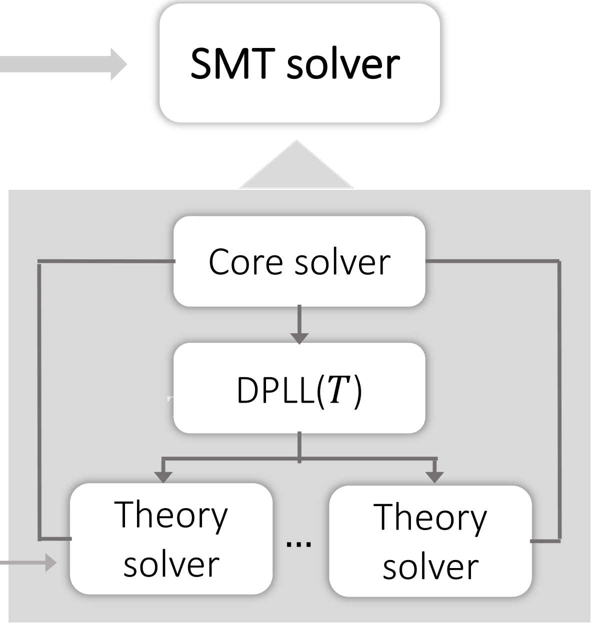
$x = f(y)$
$2x + z < 10$
$3y = 1$
⋮
$a[i] = x$
∨
∧
¬
∀
∃
SMT solver


$x = f(y)$
$2x + z < 10$
$3y = 1$
⋮
$a[i] = x$
Theories
∨
∧
¬
∀
∃
SMT solver


$x = f(y)$
$2x + z < 10$
$3y = 1$
⋮
$a[i] = x$
∨
∧
¬
∀
∃
Theories
First-order logic
SMT solver


$x = f(y)$
$2x + z < 10$
$3y = 1$
⋮
$a[i] = x$
∨
∧
¬
∀
∃
Theories
First-order logic
SMT solver
Core solver
DPLL(T)
Theory solver
…
Theory solver


$x = f(y)$
$2x + z < 10$
$3y = 1$
⋮
$a[i] = x$
∨
∧
¬
∃
∀
Theories
First-order logic
Decision procedure for checking satisfiability in quantifier-free conjunctive fragment


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

$F: x=z \land ((y=z \land x<z) \lor \neg(x=z))$

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

$A: b_1 \land b_2 \land b_3$

$B^{-1}(A) = x=z \land y=z \land x<z$

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 $\boldsymbol{T}$:

$$F := a_T^i \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F$$

- 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:

$$\mathcal{B}(a_T^i) = b_i$$

$$\mathcal{B}(F_1 \wedge F_2) = \mathcal{B}(F_1) \wedge \mathcal{B}(F_2)$$

$$\mathcal{B}(F_1 \vee F_2) = \mathcal{B}(F_1) \vee \mathcal{B}(F_2)$$

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

DPLL(T)

```
DPLL_T(F)
  G = B(F)
  while (true) do
    A, out = SAT-SOLVER(G)
    if (out = UNSAT) then return UNSAT
    else
      out = T-SOLVER (B^{-1}(A))
      if (out = SAT) then return SAT
      else G = G ∧ ¬A
```

DPLL(T)

Boolean abstraction

```
DPLL_T(F)
  G = B(F)
  while (true) do
    A, out = SAT-SOLVER(G)
    if (out = UNSAT) then return UNSAT
    else
      out = T-SOLVER (B^-1(A))
      if (out = SAT) then return SAT
      else G = G ∧ ¬A
```

DPLL(T)

```
DPLL_T(F)
  G = B(F)
  while (true) do
    A, out = SAT-SOLVER(G)
    if (out = UNSAT) then return UNSAT
    else
      out = T-SOLVER (B^{-1}(A))
      if (out = SAT) then return SAT
      else G = G ∧ ¬A
```

Boolean abstraction

conjunction of propositional literals

DPLL(T)

$\text{DPLL}_T(F)$

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

  while (true) do

    A, out = SAT-Solver(G)

    if (out = UNSAT) then return UNSAT

    else

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

      if (out = SAT) then return SAT

      else $G = G \wedge \neg A$

Boolean abstraction

conjunction of propositional literals

conjunction of atomic T-formulas

DPLL(T)

```
DPLL_T(F)
  G = B(F)
  while (true) do
    A, out = SAT-Solver(G)
    if (out = UNSAT) then return UNSAT
    else
      out = T-Solver (B^-1(A))
      if (out = SAT) then return SAT
      else G = G ∧ ¬A
```

Boolean abstraction

conjunction of propositional literals

conjunction of atomic T-formulas

theory conflict clause

DPLL(T)

```
DPLL_T(F)
  G = B(F)
  while (true) do
    A, out = SAT-Solver(G)
    if (out = UNSAT) then return UNSAT
    else
      out = T-Solver (B^{-1}(A))
      if (out = SAT) then return SAT
      else G = G ∧ ¬A
```

Boolean abstraction

conjunction of propositional literals

conjunction of atomic T-formulas

theory conflict clause

Too weak! Blocks one assignment at a time.

DPLL(T): improvement

```
DPLL_T(F)
  G = B(F)
  while (true) do
    A, out = SAT-SOLVER(G)
    if (out = UNSAT) then return UNSAT
    else
      out = T-SOLVER (B^-1(A))
      if (out = SAT) then return SAT
      else G = G ∧ ¬B(MINIMALUNSATCORE(B^-1(A)))
```

DPLL(T): improvement

```
DPLL_T(F)
  G = B(F)
  while (true) do
    A, out = SAT-SOLVER(G)
    if (out = UNSAT) then return UNSAT
    else
      out = T-SOLVER (B^{-1}(A))
      if (out = SAT) then return SAT
      else G = G ∧ ¬B(MINIMALUNSATCORE(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

DPLL(T): improvement

```
DPLL_T(F)
  G = B(F)
  while (true) do
    A, out = SAT-SOLVER(G)
    if (out = UNSAT) then return UNSAT
    else
      out = T-SOLVER (B^-1(A))
      if (out = SAT) then return SAT
      else G = G ∧ ¬B(MINIMALUNSATCORE(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 *full* assignment to the Boolean abstraction to generate conflict clause

DPLL(T): improvement

```
DPLL_T(F)
  G = B(F)
  while (true) do
    A, out = SAT-SOLVER(G)
    if (out = UNSAT) then return UNSAT
    else
      out = T-SOLVER (B^-1(A))
      if (out = SAT) then return SAT
      else G = G ∧ ¬B(MINIMALUNSATCORE(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 *full* assignment to the Boolean abstraction to generate conflict clause

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

$x = g(y)$

Core solver:
Equality and UF

$2x + y \leq 5$

Theory solver:
Linear Real Arithmetic

$2i + j \leq 5$

Theory solver:
Linear Integer Arithmetic

$(b \gg 2) = c$

Theory solver:
Fixed-Width Bitvectors

$a[i] = x$

Theory solver:
Arrays

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. $\forall x, y, z.\ (x = y \wedge y = z) \rightarrow x = z$ (transitivity)
4. $\forall x_1, \ldots, x_n, y_1, \ldots, y_n.\ (\bigwedge_i x_i = y_i) \rightarrow \big(f(x_1, \ldots \ldots, x_n) = f(y_1, \ldots .., y_n)\big)$ (fn. congruence)
5. $\forall x_1, \ldots, x_n, y_1, \ldots y_n.\ (\bigwedge_i x_i = y_i) \rightarrow ((\mathrm{p}(x_1, \ldots \ldots, x_n) \leftrightarrow p(y_1, \ldots .., y_n)$ (pr. congruence)

Theory of equality $T_=$

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. $p(x_1, \ldots, x_n) \rightsquigarrow f_p(x_1, \ldots, x_n) = \bullet$

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. $\forall x, y, z.\ (x = y \wedge y = z) \rightarrow x = z$ (transitivity)
4. $\forall x_1, \ldots, x_n, y_1, \ldots, y_n.\ (\bigwedge_i x_i = y_i) \rightarrow (f(x_1, \ldots \ldots, x_n) = f(y_1, \ldots., y_n))$ (fn. congruence)
5. $\forall x_1, \ldots, x_n, y_1, \ldots y_n.\ (\bigwedge_i x_i = y_i) \rightarrow ((p(x_1, \ldots \ldots, x_n) \leftrightarrow p(y_1, \ldots., y_n))$ (pr. congruence)

Theory of equality & uninterpreted functions $T_=$

T_{EUF}

Signature

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

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. $\forall x_1, \dots, x_n, y_1, \dots, y_n.\ (\bigwedge_i x_i = y_i) \rightarrow \big(f(x_1, \dots \dots, x_n) = f(y_1, \dots .., y_n)\big)$ (fn. congruence)

$T_=$ models

All first-order structures $\langle U, I \rangle$ that satisfy the axioms of $T_=$

Is a conjunction of $T_=$ literals satisfiable?

$$f\big(f(f(a))\big) = a \;\wedge\; f\Big(f\Big(f\big(f(f(a))\big)\Big)\Big) = a \;\wedge\; f(a) \neq a$$

Is a conjunction of $T_=$ literals satisfiable?

$$f\Big(f\big(f(a)\big)\Big) = a \quad \wedge \quad f\Bigg(f\bigg(f\Big(f\big(f(a)\big)\Big)\bigg)\Bigg) = a \quad \wedge \quad f(a) \neq a$$

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

Is a conjunction of $T_=$ literals satisfiable?

$$f\left(f\left(f(a)\right)\right) = a \quad \wedge \quad f\left(f\left(f\left(f\left(f(a)\right)\right)\right)\right) = a \quad \wedge \quad f(a) \neq a$$

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

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

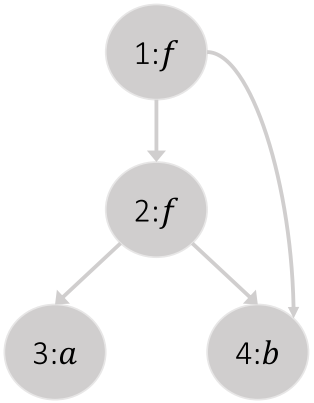
Computing the "**congruence closure**" of = over the subterm set

Congruence closure algorithm: data structure

Subterms: a, b, f(a,b), f(f(a,b))

- 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

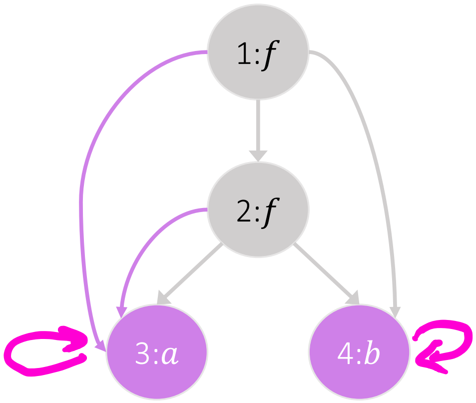
$$f(a,b) = a \ \wedge f(f(a,b),b) \neq a$$



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)

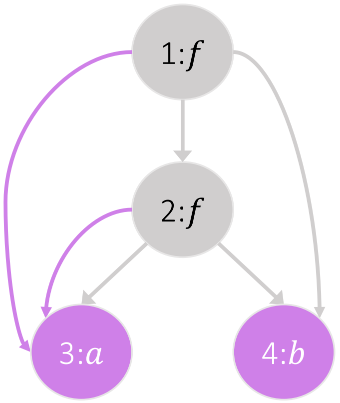
$$f(a,b) = a \wedge f(f(a,b),b) \neq a$$



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)

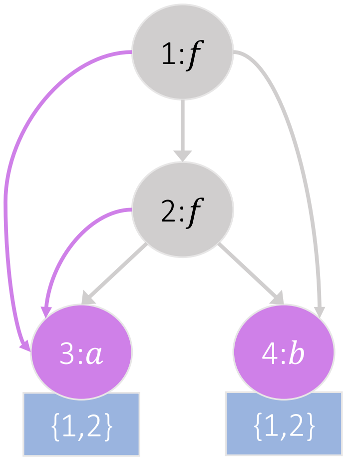
Each congruence class has one representative.
When merging two classes, only need to update the representative



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 $f(\vec{y})$



Congruence closure algorithm

```
DECIDE(F)
    construct the DAG for F's subterms
    for s_i = t_i ∈ F
        MERGE(s_i, t_i)
    for s_i ≠ t_i ∈ F
        if FIND(s_i) = FIND(t_i)
        then return UNSAT
    return SAT
```

Congruence closure algorithm

```
DECIDE(F)
    construct the DAG for F's subterms
    for s_i = t_i ∈ F
        MERGE(s_i, t_i)
    for s_i ≠ t_i ∈ 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

```
DECIDE(F)
    construct the DAG for F's subterms
    for s_i = t_i ∈ F
        MERGE(s_i, t_i)
    for s_i ≠ t_i ∈ 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

```
DECIDE(F)
    construct the DAG for F's subterms
    for s_i = t_i ∈ F
        MERGE(s_i, t_i)
    for s_i ≠ t_i ∈ 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

```
DECIDE(F)
    construct the DAG for F's subterms
    for  s_i = t_i ∈ F
       MERGE(s_i, t_i)
    for  s_i ≠ t_i ∈ 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

```
DECIDE(F)
    construct the DAG for F's subterms
    for s_i = t_i ∈ F
        MERGE(s_i, t_i)
    for s_i ≠ t_i ∈ 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

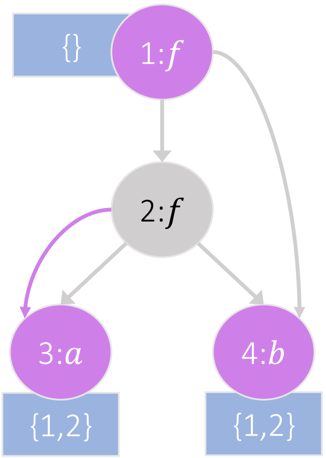
If F has a negative literal $t_1 \neq t_2$ with t_1 and t_2 in the same congruence class, output **UNSAT**

Otherwise, output **SAT**

Congruence closure algorithm: union-find

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

$$f(a,b) = a \;\wedge f(f(a,b),b) \neq a$$

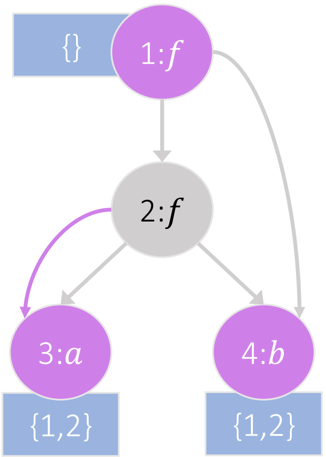


Congruence closure algorithm: union-find

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

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

FIND(2)?



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 = FIND(i_1), FIND(i_2)

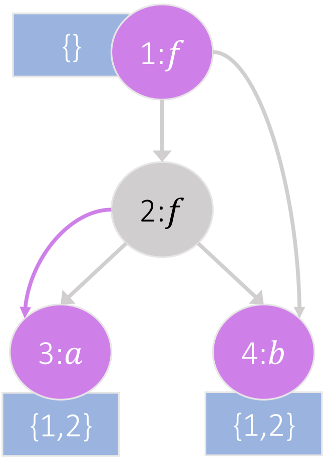
n_1.find = n_2

n_2.ccp = n_1.ccp ∪ n_2.ccp

n_1.ccp = ϕ

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

UNION(1,2)?



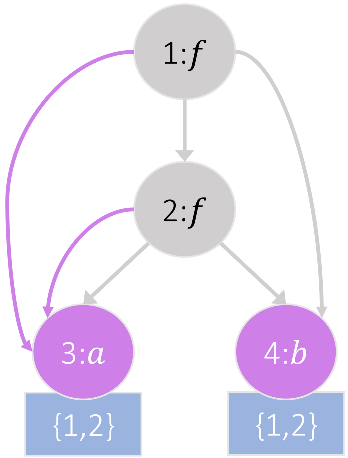
Congruence closure algorithm: union-find

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 = FIND(i_1), FIND(i_2)
n_1.find = n_2
n_2.ccp = n_1.ccp $\cup$ n_2.ccp
n_1.ccp = ϕ

$$f(a, b) = a \wedge f(f(a, b), b) \neq a$$

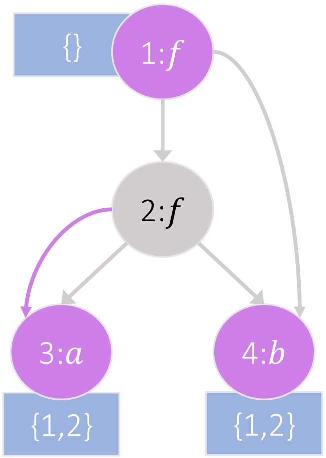


Congruence closure algorithm: congruent

CONGRUENT take as input two nodes and return true iff their:

- functions are the same
- corresponding arguments are in the same congruence class

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



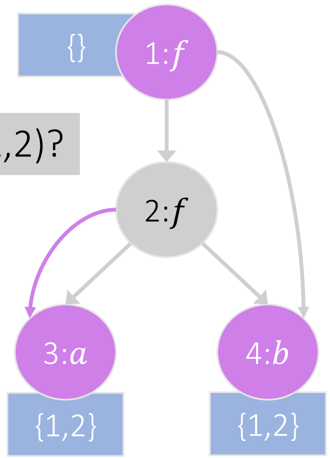
Congruence closure algorithm: congruent

CONGRUENT take as input two nodes and return true iff their:

- functions are the same
- corresponding arguments are in the same congruence class

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

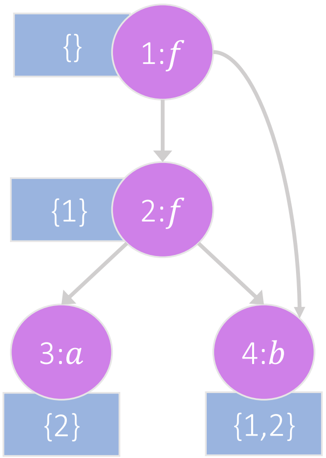
CONGRUENT(1,2)?



Congruence closure algorithm: merge

```
MERGE(i₁, i₂)
    n₁, n₂ = FIND(i₁), FIND(i₂)
    if n₁ = n₂ then return
    p₁, p₂ = n₁.ccp, n₂.ccp
    UNION(n₁, n₂)
    for each t₁, t₂ ∈ p₁ × p₂
        if FIND(t₁) ≠ FIND(t₂) ∧ CONGRUENT(t₁, t₂)
        then MERGE(t₁, t₂)
```

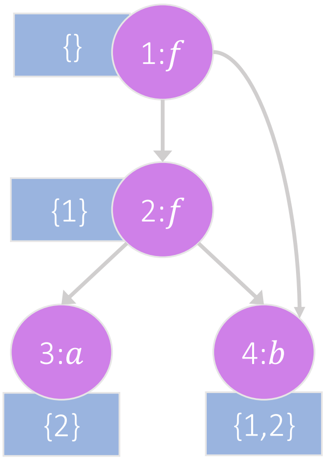
$$f(a,b) = a \ \wedge f(f(a,b),b) \neq a$$



Congruence closure algorithm: merge

```
MERGE(i1, i2)
    n1, n2 = FIND(i1), FIND(i2)
    if n1 = n2 then return
    p1, p2 = n1.ccp, n2.ccp
    UNION(n1, n2)
    for each t1, t2 ∈ p1 × p2
        if FIND(t1) ≠ FIND(t2) ∧ CONGRUENT(t1, t2)
        then MERGE(t1, t2)
```

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

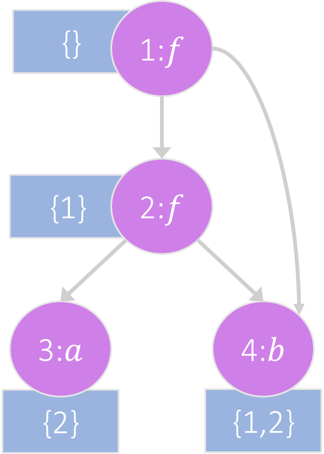


Congruence closure algorithm: merge

```
MERGE(i₁, i₂)
    n₁, n₂ = FIND(i₁), FIND(i₂)
    if n₁ = n₂ then return
    p₁, p₂ = n₁.ccp, n₂.ccp
    UNION(n₁, n₂)
    for each t₁, t₂ ∈ p₁ × p₂
        if FIND(t₁) ≠ FIND(t₂) ∧ CONGRUENT(t₁, t₂)
        then MERGE(t₁, t₂)
```

MERGE(2,3)

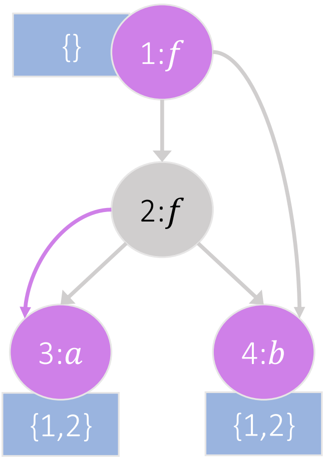
$f(a,b) = a \wedge f(f(a,b),b) \neq a$



Congruence closure algorithm: merge

```
MERGE(i₁, i₂)
    n₁, n₂ = FIND(i₁), FIND(i₂)
    if n₁ = n₂ then return
    p₁, p₂ = n₁.ccp, n₂.ccp
    UNION(n₁, n₂)
    for each t₁, t₂ ∈ p₁ × p₂
        if FIND(t₁) ≠ FIND(t₂) ∧ CONGRUENT(t₁, t₂)
        then MERGE(t₁, t₂)
```

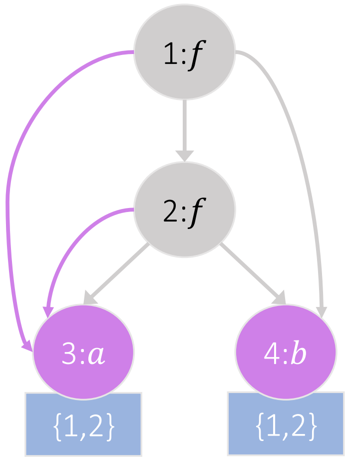
$f(a,b) = a \wedge f(f(a,b),b) \neq a$



Congruence closure algorithm: merge

```
MERGE(i₁, i₂)
    n₁, n₂ = FIND(i₁), FIND(i₂)
    if n₁ = n₂ then return
    p₁, p₂ = n₁.ccp, n₂.ccp
    UNION(n₁, n₂)
    for each t₁, t₂ ∈ p₁ × p₂
        if FIND(t₁) ≠ FIND(t₂) ∧ CONGRUENT(t₁, t₂)
        then MERGE(t₁, t₂)
```

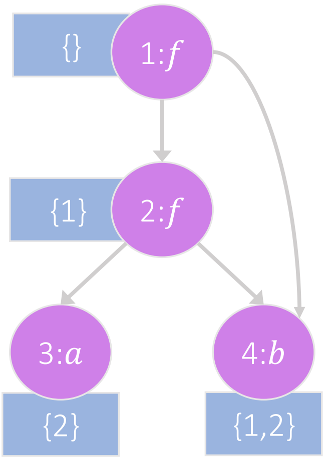
$f(a, b) = a \land f(f(a, b), b) \neq a$



Congruence closure algorithm

```
DECIDE(F)
    construct the DAG for F's subterms
    for s_i = t_i ∈ F
        MERGE(s_i, t_i)
    for s_i ≠ t_i ∈ F
        if FIND(s_i) = FIND(t_i)
        then return UNSAT
    return SAT
```

$$f(a,b) = a \ \wedge f(f(a,b),b) \neq a$$



Congruence closure algorithm

DECIDE(F)

construct the DAG for F's subterms

for $s_i = t_i \in F$

MERGE(s_i, t_i)

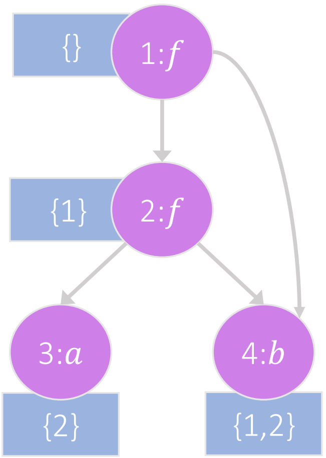
for $s_i \neq t_i \in F$

if FIND(s_i) = FIND(t_i)

then return UNSAT

return SAT

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



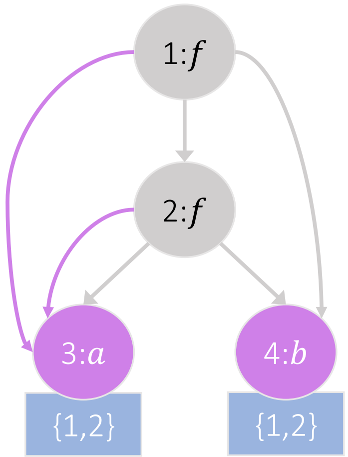
Congruence closure algorithm

DECIDE(F)

construct the DAG for F's subterms

```
for s_i = t_i ∈ F
    MERGE(s_i, t_i)
for s_i ≠ t_i ∈ F
    if FIND(s_i) = FIND(t_i)
    then return UNSAT
return SAT
```

$f(a,b) = a \wedge f(f(a,b),b) \neq a$



Congruence closure algorithm

DECIDE(F)

construct the DAG for F's subterms

for $s_i = t_i \in F$

MERGE(s_i, t_i)

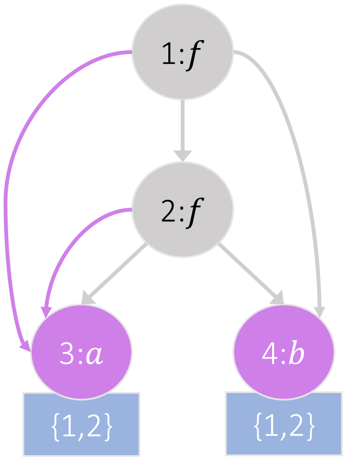
for $s_i \neq t_i \in F$

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

DECIDE(F)

construct the DAG for F's subterms

for $s_i = t_i \in F$

MERGE(s_i, t_i)

for $s_i \neq t_i \in F$

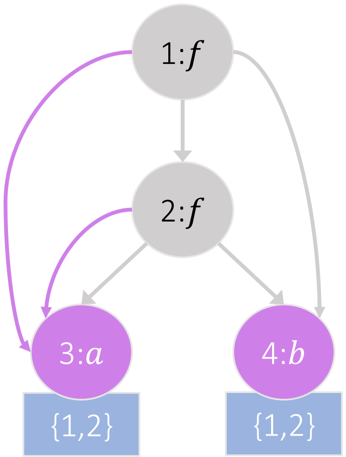
if FIND(s_i) = FIND(t_i)

then return UNSAT

return SAT

$$f(a,b) = a \wedge f(f(a,b),b) \neq a$$

UNSAT!



Definitions I

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: $\forall s_1, s_2, s_3 \in S.\ R(s_1,s_2) \wedge R(s_2,s_3) \rightarrow R(s_1,s_3)$

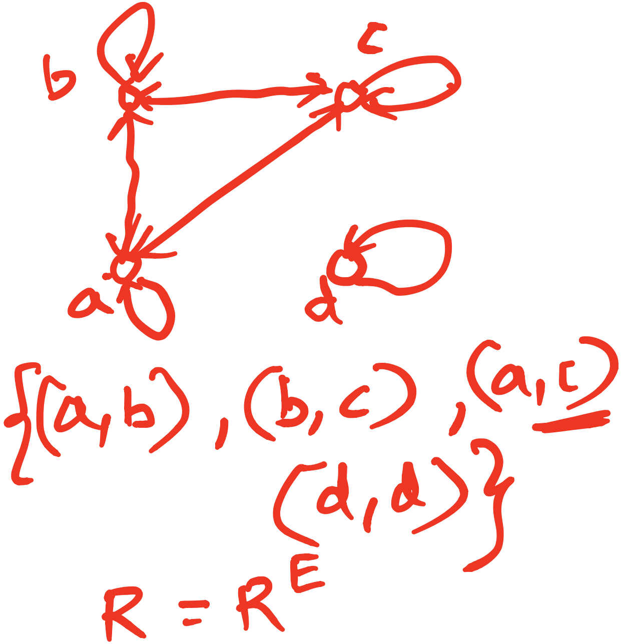
The **equivalence class** of element $s \in S$ under R: $[s]_R \stackrel{\text{def}}{=} \{s' \in S : R(s,s')\}$

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

Definitions II



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 $R^C \prec R'$

Definitions II

R^E is the smallest equivalence relation that includes R.

R^C is the smallest congruence relation that includes 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 $\sum_=$ formula as follows:
$s_1 = t_1 \wedge \ldots \wedge s_m = t_m \wedge s_{m+1} \neq t_{m+1} \wedge \ldots \wedge s_n \neq t_n$

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 $\sum_{=}$ formula as follows:
$s_1 = t_1 \wedge \ldots \wedge s_m = t_m \wedge s_{m+1} \neq t_{m+1} \wedge \ldots \wedge s_n \neq t_n$

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 $[m+1, n]$, $s_i \nsim t_i$

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

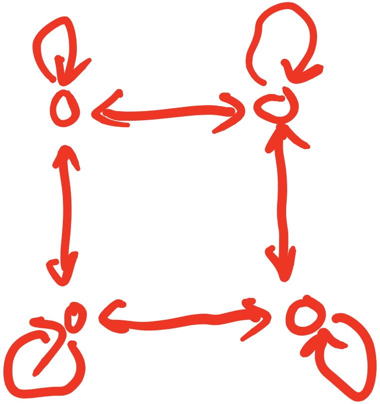
Summary

Today

- SMT solving
- DPLL(T) : combine DPLL algorithm for SAT solving with theory solvers
- A core theory solver: congruence closure algorithm for $\boldsymbol{T}_=$

Next

- Temporal logic



R_1

R_2

$$R_1 = R_1^E < R_1 \cup R_2$$

$$R_2 = R_2^E \{ R_1 \cup R_2$$