On Designing and Implementing Satisfiability Modulo Theory (SMT) Solvers

Summer School 2009, Nancy
Verification Technology, Systems and Applications

Leonardo de Moura
Microsoft Research
Verification/Analysis tools need some form of Symbolic Reasoning
Logic is “The Calculus of Computer Science” (Z. Manna).

High computational complexity
Applications

- Test case generation
- Verifying Compilers
- Predicate Abstraction
- Invariant Generation
- Type Checking
- Model Based Testing
Some Applications @ Microsoft

- VCC
- Hyper-V
- HAVOC
- Terminator T-2
- VCC
- NModel
- SpecExplorer
- SAGE
- SLAM
- Vigilante
- F7
unsigned GCD(x, y) {
    requires(y > 0);
    while (true) {
        unsigned m = x % y;
        if (m == 0) return y;
        x = y;
        y = m;
    }
}

We want a trace where the loop is executed twice.

(y₀ > 0) and
(m₀ = x₀ % y₀) and
not (m₀ = 0) and
(x₁ = y₀) and
(y₁ = m₀) and
(m₁ = x₁ % y₁) and
(m₁ = 0)

x₀ = 2
y₀ = 4
m₀ = 2
x₁ = 4
y₁ = 2
m₁ = 0
Type checking

Signature:
\[ \text{div} : \text{int}, \{ x : \text{int} \mid x \neq 0 \} \rightarrow \text{int} \]

Call site:
if \( a \leq 1 \) and \( a \leq b \) then
return \( \text{div}(a, b) \)

Verification condition
\( a \leq 1 \) and \( a \leq b \) implies \( b \neq 0 \)
Is formula $F$ satisfiable modulo theory $T$?

SMT solvers have specialized algorithms for $T$. 
Satisfiability Modulo Theories (SMT)

\[ b + 2 = c \text{ and } f(\text{read}(\text{write}(a,b,3),c-2)) \neq f(c-b+1) \]
Satisfiability Modulo Theories (SMT)

\[ b + 2 = c \quad \text{and} \quad f(\text{read}(\text{write}(a,b,3),c-2)) \neq f(c-b+1) \]
$b + 2 = c \text{ and } f(\text{read}(\text{write}(a, b, 3), c-2)) \neq f(c-b+1)$

Array Theory
b + 2 = c and $f(\text{read(\text{write}(a,b,3), c-2))) \neq f(\text{c-b+1})$
\[ b + 2 = c \quad \text{and} \quad f(\text{read}(\text{write}(a,b,3), c-2)) \neq f(c-b+1) \]

Substituting \( c \) by \( b+2 \)
b + 2 = c and f(read(write(a,b,3), b+2-2)) ≠ f(b+2-b+1)

Simplifying
b + 2 = c and $f(\text{read}(\text{write}(a,b,3), b)) \neq f(3)$
b + 2 = c and \( f(\text{read}(\text{write}(a, b, 3), b)) \neq f(3) \)

Applying array theory axiom
for all \( a, i, v \): \( \text{read}(\text{write}(a, i, v), i) = v \)
b + 2 = c and f(3) ≠ f(3)

Inconsistent/Unsatisfiable
Repository of Benchmarks

http://www.smtlib.org

Benchmarks are divided in “logics”:

- QF_UF: unquantified formulas built over a signature of uninterpreted sort, function and predicate symbols.
- QF_UFLIA: unquantified linear integer arithmetic with uninterpreted sort, function, and predicate symbols.
- AUFLIA: closed linear formulas over the theory of integer arrays with free sort, function and predicate symbols.
For most SMT solvers: \textit{F is a set of ground formulas}

Many Applications

Bounded Model Checking

Test-Case Generation
An SMT Solver is a collection of Little Engines of Proof
An SMT Solver is a collection of Little Engines of Proof

Examples:
SAT Solver (Daniel’s lectures)
Equality solver
a = b, b = c, d = e, b = s, d = t, a \ne e, a \ne s
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s\]

\[a, b \quad c \quad d \quad e \quad s \quad t\]
Deciding Equality

\[ a = b, \quad b = c, \quad d = e, \quad b = s, \quad d = t, \quad a \neq e, \quad a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
a = b, b = c, d = e, \( b = s \), d = t, \( a \neq e \), \( a \neq s \)
a = b, b = c, d = e, b = s, d = t, a \neq e, a \neq s
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

a = b, b = c, d = e, b = s, d = t, a ≠ e, a ≠ s

Unsatisfiable
Deciding Equality

\[a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e\]

Model construction
a = b, b = c, d = e, b = s, d = t, a ≠ e

Model construction

|M| = {♦₁, ♦₂} (universe, aka domain)
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e \]

Model construction

\[ |M| = \{\blacklozenge_1, \blacklozenge_2\} \] (universe, aka domain)

\[ M(a) = \blacklozenge_1 \] (assignment)
a = b, b = c, d = e, b = s, d = t, a ≠ e

Model construction

\(|M| = \{\clubsuit_1, \clubsuit_2\} \) (universe, aka domain)
M(a) = \clubsuit_1 (assignment)
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e \]

Model construction

\[ |M| = \{\diamond_1, \diamond_2\} \text{ (universe, aka domain)} \]

\[ \begin{align*}
M(a) &= M(b) = M(c) = M(s) = \diamond_1 \\
M(d) &= M(e) = M(t) = \diamond_2
\end{align*} \]
Deciding Equality: Termination, Soundness, Completeness

- **Termination**: easy

- **Soundness**
  - Invariant: all constants in a “ball” are known to be equal.
  - The “ball” merge operation is justified by:
    - Transitivity and Symmetry rules.

- **Completeness**
  - We can build a model if an inconsistency was not detected.
  - Proof template (by contradiction):
    - Build a candidate model.
    - Assume a literal was not satisfied.
    - Find contradiction.
Completeness

We can build a model if an inconsistency was not detected.

Instantiating the template for our procedure:

Assume some literal $c = d$ is not satisfied by our model.

That is, $M(c) \neq M(d)$.

This is impossible, $c$ and $d$ must be in the same “ball”.

\[ M(c) = M(d) = \bigcirc_i \]
Completeness

We can build a model if an inconsistency was not detected.

Instantiating the template for our procedure:

- Assume some literal \( c \neq d \) is not satisfied by our model.
- That is, \( M(c) = M(d) \).
- Key property: we only check the disequalities after we processed all equalities.
- This is impossible, \( c \) and \( d \) must be in the different “balls”

\[ M(c) = \Diamond_i \]
\[ M(d) = \Diamond_j \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e)) \]

**Congruence Rule:**

\[ x_1 = y_1, \ldots, x_n = y_n \implies f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e)) \]

First Step: “Naming” subterms

Congruence Rule:
\[ x_1 = y_1, ..., x_n = y_n \text{ implies } f(x_1, ..., x_n) = f(y_1, ..., y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, v_1) \neq f(b, g(e)) \]
\[ v_1 \equiv g(d) \]

First Step: “Naming” subterms

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, \ v_1) \neq f(b, \ g(e)) \]
\[ v_1 \equiv g(d) \]

First Step: “Naming” subterms

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[
a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, v_1) \neq f(b, v_2)
\]
\[
v_1 \equiv g(d), \ v_2 \equiv g(e)
\]

First Step: “Naming” subterms

Congruence Rule:
\[
x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)
\]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, v_1) \neq f(b, v_2) \]
\[ v_1 \equiv g(d), \ v_2 \equiv g(e) \]

First Step: “Naming” subterms

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ v_3 \neq f(b, v_2) \]
\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1) \]

First Step: “Naming” subterms

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ \begin{align*}
a &= b, & b &= c, & d &= e, & b &= s, & d &= t, & v_3 &\neq f(b, v_2) \\
v_1 &\equiv g(d), & v_2 &\equiv g(e), & v_3 &\equiv f(a, v_1) \\
\end{align*} \]

First Step: “Naming” subterms

Congruence Rule:

\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ v_3 \neq v_4 \]
\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

First Step: “Naming” subterms

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ v_3 \neq v_4 \]

\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

Congruence Rule:

\[ x_1 = y_1, \ldots, x_n = y_n \implies f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[
a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ v_3 \neq v_4
\]

\[
v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2)
\]

**Congruence Rule:**

\[
x_1 = y_1, \ ... , \ x_n = y_n \text{ implies } f(x_1, ... , x_n) = f(y_1, ... , y_n)
\]

\[
d = e \text{ implies } g(d) = g(e)
\]

a, b, c, s  

d, e, t  

v_1  

v_2  

v_3  

v_4  

Microsoft Research
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ v_3 \neq v_4 \]

\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

**Congruence Rule:**

\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]

\[ d = e \text{ implies } v_1 = v_2 \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ v_1 \neq v_4 \]

\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

\( v_1 \) and \( v_2 \) are congruent.

\[ x_1 = y_1, \ldots, x_n = y_n \implies f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]

\( d = e \implies v_1 = v_2 \)
Deciding Equality + (uninterpreted) Functions

\[ a = b, \quad b = c, \quad d = e, \quad b = s, \quad d = t, \quad v_3 \neq v_4 \]

\[ v_1 \equiv g(d), \quad v_2 \equiv g(e), \quad v_3 \equiv f(a, v_1), \quad v_4 \equiv f(b, v_2) \]

Congruence Rule:

\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]

\[ a = b, \quad v_1 = v_2 \text{ implies } f(a, v_1) = f(b, v_2) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ v_3 \neq v_4 \]

\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

Congruence Rule:

\[ x_1 = y_1, \ldots, x_n = y_n \implies f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]

\[ a = b, \ v_1 = v_2 \implies v_3 = v_4 \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ v_3 \not\equiv v_4 \]
\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
\[ a = b, \ v_1 = v_2 \text{ implies } v_3 = v_4 \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ v_3 \neq v_4 \]
\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

Congruence Rule:
\[ x_1 = y_1, \ ..., \ x_n = y_n \text{ implies } f(x_1, ..., x_n) = f(y_1, ..., y_n) \]

Unsatisfiable
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ \ a \neq v_4, \ v_2 \neq v_3 \]
\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

Changing the problem

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq v_4, \ v_2 \neq v_3 \]
\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \implies f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq v_4, \ v_2 \neq v_3 \]
\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

Congruence Rule:
\[ x_1 = y_1, \ ... , \ x_n = y_n \text{ implies } f(x_1, ... , x_n) = f(y_1, ... , y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq v_4, \ v_2 \neq v_3 \]

\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

Model construction:

\[ |M| = \{ \downarrow_1, \downarrow_2, \downarrow_3, \downarrow_4 \} \]

\[ M(a) = M(b) = M(c) = M(s) = \downarrow_1 \]

\[ M(d) = M(e) = M(t) = \downarrow_2 \]

\[ M(v_1) = M(v_2) = \downarrow_3 \]

\[ M(v_3) = M(v_4) = \downarrow_4 \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \; b = c, \; d = e, \; b = s, \; d = t, \; a \neq v_4, \; v_2 \neq v_3 \]
\[ v_1 \equiv g(d), \; v_2 \equiv g(e), \; v_3 \equiv f(a, v_1), \; v_4 \equiv f(b, v_2) \]

Model construction:

\[ |M| = \{,\}, M(a) = M(b) = M(c) = M(s) = \}
M(d) = M(e) = M(t) = \}
M(v_1) = M(v_2) = \}
M(v_3) = M(v_4) = \}

Missing: Interpretation for f and g.
Deciding Equality + (uninterpreted) Functions

Building the interpretation for function symbols

- M(g) is a mapping from |M| to |M|
- Defined as:
  
  \[ M(g)(\Diamond_i) = \Diamond_j \text{ if there is } v \equiv g(a) \text{ s.t.} \]
  
  \[
  \begin{align*}
  M(a) &= \Diamond_i \\
  M(v) &= \Diamond_j \\
  = \Diamond_k, \text{ otherwise (} \Diamond_k \text{ is an arbitrary element)}
  \end{align*}
  \]

- Is M(g) well-defined?
Deciding Equality + (uninterpreted) Functions

Building the interpretation for function symbols

- \( M(g) \) is a mapping from \(|M|\) to \(|M|\)
- Defined as:
  \[
  M(g)(\diamond_i) = \diamond_j \quad \text{if there is } v \equiv g(a) \text{ s.t.}
  \]
  \[
  M(a) = \diamond_i \quad M(v) = \diamond_j \quad M(\diamond_k), \text{ otherwise (\diamond_k is an arbitrary element)}
  \]
- Is \( M(g) \) well-defined?
- Problem: we may have \( v \equiv g(a) \) and \( w \equiv g(b) \) s.t.
  \[
  M(a) = M(b) = \diamond_1 \quad \text{and } M(v) = \diamond_2 \neq \diamond_3 = M(w)
  \]
  So, is \( M(g)(\diamond_1) = \diamond_2 \) or \( M(g)(\diamond_1) = \diamond_3 \)?
Building the interpretation for function symbols

- $M(g)$ is a mapping from $|M|$ to $|M|$.

Defined as:

$$M(g)(\bullet_i) = \bullet_j \text{ if there is } v \equiv g(a) \text{ such that }$$

- $M(a) = \bullet_i$
- $M(v) = \bullet_j$

$$= \bullet_k, \text{ otherwise (} \bullet_k \text{ is an arbitrary element)}$$

- Is $M(g)$ well-defined?

Problem: we may have $v \equiv g(a)$ and $w \equiv g(b)$ s.t.

- $M(a) = M(b) = \bullet_1$
- $M(v) = \bullet_2 \neq \bullet_3 = M(w)$

So, is $M(g)(\bullet_1) = \bullet_2$ or $M(g)(\bullet_1) = \bullet_3$?

This is impossible because of the congruence rule!

- $a$ and $b$ are in the same “ball”, then so are $v$ and $w$
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq v_4, \ v_2 \neq v_3 \]
\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

Model construction:

\[ |M| = \{ v_1, v_2, v_3, v_4 \} \]
\[ M(a) = M(b) = M(c) = M(s) = v_1 \]
\[ M(d) = M(e) = M(t) = v_2 \]
\[ M(v_1) = M(v_2) = v_3 \]
\[ M(v_3) = M(v_4) = v_4 \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq v_4, \ v_2 \neq v_3 \]
\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, v_1), \ v_4 \equiv f(b, v_2) \]

Model construction:

\[ |M| = \{\blacklozenge_1, \blacklozenge_2, \blacklozenge_3, \blacklozenge_4\} \]

\[ M(a) = M(b) = M(c) = M(s) = \blacklozenge_1 \]
\[ M(d) = M(e) = M(t) = \blacklozenge_2 \]
\[ M(v_1) = M(v_2) = \blacklozenge_3 \]
\[ M(v_3) = M(v_4) = \blacklozenge_4 \]

\[ M(g)(\blacklozenge_i) = \blacklozenge_j \text{ if there is } v \equiv g(a) \text{ s.t.} \]
\[ M(a) = \blacklozenge_i \]
\[ M(v) = \blacklozenge_j \]
\[ = \blacklozenge_k, \text{ otherwise} \]
Deciding Equality + (uninterpreted) Functions

\[a = b, b = c, d = e, b = s, d = t, a \neq v_4, v_2 \neq v_3\]

\[v_1 \equiv g(d), v_2 \equiv g(e), v_3 \equiv f(a, v_1), v_4 \equiv f(b, v_2)\]

Model construction:

\[|M| = \{\uparrow_1, \uparrow_2, \uparrow_3, \uparrow_4\}\]

\[M(a) = M(b) = M(c) = M(s) = \uparrow_1\]

\[M(d) = M(e) = M(t) = \uparrow_2\]

\[M(v_1) = M(v_2) = \uparrow_3\]

\[M(v_3) = M(v_4) = \uparrow_4\]

\[M(g) = \{\uparrow_2 \rightarrow \uparrow_3\}\]

\[M(g)(\uparrow_i) = \uparrow_j\] if there is \(v \equiv g(a)\) s.t.

\[M(a) = \uparrow_i\]

\[M(v) = \uparrow_j\]

\[= \uparrow_k, \text{ otherwise}\]
Deciding Equality + (uninterpreted) Functions

a = b, b = c, d = e, b = s, d = t, a \neq v_4, v_2 \neq v_3
v_1 \equiv g(d), v_2 \equiv g(e), v_3 \equiv f(a, v_1), v_4 \equiv f(b, v_2)

Model construction:

\[ |M| = \{ \diamondsuit_1, \diamondsuit_2, \diamondsuit_3, \diamondsuit_4 \} \]
\[ M(a) = M(b) = M(c) = M(s) = \diamondsuit_1 \]
\[ M(d) = M(e) = M(t) = \diamondsuit_2 \]
\[ M(v_1) = M(v_2) = \diamondsuit_3 \]
\[ M(v_3) = M(v_4) = \diamondsuit_4 \]
\[ M(g) = \{ \diamondsuit_2 \rightarrow \diamondsuit_3 \} \]

M(g)(\diamondsuit_i) = \diamondsuit_j \text{ if there is } v \equiv g(a) \text{ s.t.}
\[ M(a) = \diamondsuit_i \]
\[ M(v) = \diamondsuit_j \]
\[ = \diamondsuit_k, \text{ otherwise} \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, b = c, d = e, b = s, d = t, a \neq v_4, v_2 \neq v_3 \]
\[ v_1 \equiv g(d), v_2 \equiv g(e), v_3 \equiv f(a, v_1), v_4 \equiv f(b, v_2) \]

Model construction:

\[ |M| = \{ \bullet_1, \bullet_2, \bullet_3, \bullet_4 \} \]
\[ M(a) = M(b) = M(c) = M(s) = \bullet_1 \]
\[ M(d) = M(e) = M(t) = \bullet_2 \]
\[ M(v_1) = M(v_2) = \bullet_3 \]
\[ M(v_3) = M(v_4) = \bullet_4 \]
\[ M(g) = \{ \bullet_2 \rightarrow \bullet_3, \text{else } \rightarrow \bullet_1 \} \]

\[ M(g)(\bullet_i) = \bullet_j \text{ if there is } v \equiv g(a) \text{ s.t.} \]
\[ M(a) = \bullet_i \]
\[ M(v) = \bullet_j \]
\[ = \bullet_k, \text{ otherwise} \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq v_4, \ v_2 \neq v_3 \]

\[ v_1 \equiv g(d), \ v_2 \equiv g(e), \ v_3 \equiv f(a, \ v_1), \ v_4 \equiv f(b, \ v_2) \]

Model construction:

\[ |M| = \{ \spadesuit_1, \spadesuit_2, \spadesuit_3, \spadesuit_4 \} \]

\[ M(a) = M(b) = M(c) = M(s) = \spadesuit_1 \]

\[ M(d) = M(e) = M(t) = \spadesuit_2 \]

\[ M(v_1) = M(v_2) = \spadesuit_3 \]

\[ M(v_3) = M(v_4) = \spadesuit_4 \]

\[ M(g)(\spadesuit_i) = \spadesuit_j \text{ if there is } v \equiv g(a) \text{ s.t.} \]

\[ M(a) = \spadesuit_i \]

\[ M(v) = \spadesuit_j \]

\[ = \spadesuit_k, \text{ otherwise} \]

\[ M(f) = \{ (\spadesuit_1, \spadesuit_3) \rightarrow \spadesuit_4, \text{ else } \rightarrow \spadesuit_1 \} \]
Deciding Equality + (uninterpreted) Functions

What about predicates?

\[ p(a, b), \neg p(c, b) \]
What about predicates?

\[ p(a, b), \quad \neg p(c, b) \]

\[ f_p(a, b) = T, \quad f_p(c, b) \neq T \]
It is possible to eliminate function symbols using a method called **Ackermannization**.

\[
a = b, \quad b = c, \quad d = e, \quad b = s, \quad d = t, \quad a \neq v_4, \quad v_2 \neq v_3
\]

\[
v_1 \equiv g(d), \quad v_2 \equiv g(e), \quad v_3 \equiv f(a, v_1), \quad v_4 \equiv f(b, v_2)
\]

\[
a = b, \quad b = c, \quad d = e, \quad b = s, \quad d = t, \quad a \neq v_4, \quad v_2 \neq v_3
\]

\[
d \neq e \lor v_1 = v_2,
\]

\[
a \neq v_1 \lor b \neq v_2 \lor v_3 = v_4
\]
It is possible to eliminate function symbols using a method called **Ackermannization**.

\[
\begin{align*}
a &= b, \ b &= c, \ d &= e, \ b &= s, \ d &= t, \ a &\neq v_4, \ v_2 &\neq v_3 \\
v_1 &\equiv g(d), \ v_2 &\equiv g(e), \ v_3 &\equiv f(a, v_1), \ v_4 &\equiv f(b, v_2)
\end{align*}
\]

\[
\begin{align*}
a &= b, \ b &= c, \ d &= e, \ b &= s, \ d &= t, \ a &\neq v_4, \ v_2 &\neq v_3 \\
d &\neq e \lor v_1 = v_2, \\
a &\neq v_1 \lor b &\neq v_2 \lor v_3 = v_4
\end{align*}
\]

**Main Problem:** quadratic blowup
Deciding Equality + (uninterpreted) Functions

It is possible to implement our procedure in $O(n \log n)$
Deciding Equality + (uninterpreted) Functions

Sets (equivalence classes)

Union

Membership
Deciding Equality + (uninterpreted) Functions

Sets (equivalence classes)

$$d, e, t \cup t = d, e, t$$

Key observation:
The sets are disjoint!

Union

Membership

$$a, b, c, s \neq s$$
Union-Find data-structure

Every set (equivalence class) has a root element (representative).

We say: find[c] is b
Deciding Equality + (uninterpreted) Functions

Union-Find data-structure

\[ a, b, c \cup s, r = a, b, c, s, r \]
Deciding Equality + (uninterpreted) Functions

Tracking the equivalence classes size is important!

\[ a_1 \rightarrow a_2 \quad \cup \quad a_3 \quad = \quad a_1 \rightarrow a_2 \rightarrow a_3 \]

\[ a_1 \rightarrow a_2 \rightarrow a_3 \quad \cup \quad a_4 \quad = \quad a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \]

... 

\[ a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \ldots \rightarrow a_{n-1} \quad \cup \quad a_n \quad = \quad a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \ldots \rightarrow a_{n-1} \rightarrow a_n \]
Deciding Equality + (uninterpreted) Functions

Tracking the equivalence classes size is important!

\[ a_1 \rightarrow a_2 \cup a_3 = a_1 \rightarrow a_2 \leftarrow a_3 \]

\[ a_1 \rightarrow a_2 \leftarrow a_3 \cup a_4 = a_1 \rightarrow a_2 \leftarrow a_3 \]

... 

\[ a_2 \quad \cup \quad a_n = a_2 \leftarrow a_n \]

\[ \quad a_1 \quad a_3 \quad a_{n-1} \]
Deciding Equality + (uninterpreted) Functions

Tracking the equivalence classes size is important!

We can do \( n \) merges in \( O(n \log n) \)

Each constant has two fields: find and size.
Implementing the congruence rule.

Occurrences of a constant: we say $a$ occurs in $v$ iff $v \equiv f(...,a,...)$

When we “merge” two equivalence classes we can traverse these occurrences to find new congruences.

$\text{occurrences}[b] = \{ v_1 \equiv g(b), \; v_2 \equiv f(a) \}$

$\text{occurrences}[s] = \{ v_3 \equiv f(r) \}$
Implementing the congruence rule.

Occurrences of a constant: we say $a$ occurs in $v$ iff $v \equiv f(\ldots, a, \ldots)$

When we “merge” two equivalence classes we can traverse these occurrences to find new congruences.

\[
\text{occurrences}(b) = \{ v_1 \equiv g(b), v_2 \equiv f(a) \} \\
\text{occurrences}(s) = \{ v_3 \equiv f(r) \}
\]

Inefficient version:
for each $v$ in $\text{occurrences}(b)$
for each $w$ in $\text{occurrences}(s)$
if $v$ and $w$ are congruent
add $(v, w)$ to todo queue

A queue of pairs that need to be merged.
Deciding Equality + (uninterpreted) Functions

\[
\begin{align*}
\text{occurrences}[b] &= \{ v_1 \equiv g(b), v_2 \equiv f(a) \} \\
\text{occurrences}[s] &= \{ v_3 \equiv f(r) \}
\end{align*}
\]

We also need to merge \( \text{occurrences}[b] \) with \( \text{occurrences}[s] \). This can be done in constant time:

Use circular lists to represent the occurrences. (More later)
Avoiding the nested loop:
for each v in occurrences[b]
    for each w in occurrences[s]
        ...

Use a hash table to store the elements $v_1 \equiv f(a_1, \ldots, a_n)$. Each constant has an identifier (e.g., natural number). Compute hash code using the identifier of the (equivalence class) roots of the arguments.

$hash(v_1) = hash\text{-}tuple(id(f), id(root(a_1)), \ldots, id(root(a_n)))$
Deciding Equality + (uninterpreted) Functions

Avoiding the nested loop:
for each \( v \) in occurrences(b)
  for each \( w \) in occurrences(s)
    ...

Use a hash table to store \( v \) = \( f(a_1, \ldots, a_n) \).
Each constant has an identifier (e.g., natural number).
Compute hash code using the identifier of the (equivalence class) roots of the arguments.

\[
\text{hash}(v_1) = \text{hash-tuple}(\text{id}(f), \text{id}(\text{root}(a_1)), \ldots, \text{id}(\text{root}(a_n)))
\]

hash-tuple can be the Jenkin’s hash function for strings. Just adding the ids produces a very bad hash-code!
Deciding Equality + (uninterpreted) Functions

Efficient implementation of the congruence rule.
Merging the equivalences classes with roots: \( a_1 \) and \( a_2 \)
Assume \( a_2 \) is smaller than \( a_1 \)

**Before merging the equivalence classes: \( a_1 \) and \( a_2 \)**
for each \( v \) in occurrences[\( a_2 \)]

- remove \( v \) from the hash table (its hashcode will change)

**After merging the equivalence classes: \( a_1 \) and \( a_2 \)**
for each \( v \) in occurrences[\( a_2 \)]

- if there is \( w \) congruent to \( v \) in the hash-table
  - add \((v,w)\) to todo queue
- else add \( v \) to hash-table
Deciding Equality + (uninterpreted) Functions

Efficient implementation of the congruence rule.

Merging the equivalences classes with roots $a_1$ and $a_2$

Assume $a_2$ is smaller than $a_1$

Before merging the equivalence classes: $a_1$ and $a_2$

for each $v$ in occurrences[$a_2$]

    remove $v$ from the hash table (its hashcode will change)

After merging the equivalence classes: $a_1$ and $a_2$

for each $v$ in occurrences[$a_2$]

    if there is $w$ congruent to $v$ in the hash-table
        add $(v,w)$ to todo queue
    else add $v$ to hash-table

    add $v$ to occurrences($a_1$)

Trick:
Use dynamic arrays to represent the occurrences
Deciding Equality + (uninterpreted) Functions

The efficient version is not optimal (in theory).
Problem: we may have $v \equiv f(a_1, ..., a_n)$ with “huge” $n$.

Solution: currying
Use only binary functions, and represent $f(a_1, a_2, a_3, a_4)$ as 
$f(a_1, h(a_2, h(a_3, a_4)))$

This is not necessary in practice, since the $n$ above is small.
Each constant has now three fields: find, size, and occurrences.

We also has use a hash-table for implementing the congruence rule.

We will need many more improvements!
Many verification/analysis problems require: case-analysis

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]
Many verification/analysis problems require: case-analysis

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Naïve Solution: Convert to DNF

\[ (x \geq 0, \ y = x + 1, \ y > 2) \lor (x \geq 0, \ y = x + 1, \ y < 1) \]
Many verification/analysis problems require:

**case-analysis**

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

**Naïve Solution: Convert to DNF**

\[ (x \geq 0, \ y = x + 1, \ y > 2) \lor (x \geq 0, \ y = x + 1, \ y < 1) \]

Too Inefficient!
(exponential blowup)
SMT: Basic Architecture

SAT + Theory Solvers = SMT

- Equality + UF
- Arithmetic
- Bit-vectors
- ...

Case Analysis
SAT (propositional checkers): Case Analysis

\[ p \lor q, \]
\[ p \lor \neg q, \]
\[ \neg p \lor q, \]
\[ \neg p \lor \neg q \]
SAT (propositional checkers): Case Analysis

\[ p \lor q, \]
\[ p \lor \neg q, \]
\[ \neg p \lor q, \]
\[ \neg p \lor \neg q \]

Assignment:
\[ p = \text{false}, \]
\[ q = \text{false} \]
SAT (propositional checkers): Case Analysis

Assignment:
\[ p = \text{false}, \]
\[ q = \text{true} \]
SAT (propositional checkers): Case Analysis

Assignment:

\[ p = \text{true}, \]
\[ q = \text{false} \]
SAT (propositional checkers): Case Analysis

Assignment:
\[ p = true, \quad q = true \]
DPLL

Partial model

Set of clauses

M | F
Guessing

\[ p \mid p \lor q, \neg q \lor r \]

\[ p, \neg q \mid p \lor q, \neg q \lor r \]
Deducing

\[ p \mid p \lor q, \neg p \lor s \]

\[ p, s \mid p \lor q, \neg p \lor s \]
Backtracking

$p, \neg s, q \mid p \lor q, s \lor q, \neg p \lor \neg q$

$p, s \mid p \lor q, s \lor q, \neg p \lor \neg q$
Modern DPLL

- Efficient indexing (two-watch literal)
- Non-chronological backtracking (backjumping)
- Lemma learning

...
Basic Idea

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]
\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]
SAT + Theory solvers

**Basic Idea**

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]
SAT + Theory solvers

Basic Idea

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]
\[ p_1 : (x \geq 0), \ p_2 : (y = x + 1), \]
\[ p_3 : (y > 2), \ p_4 : (y < 1) \]

SAT Solver

Assignment

\[ p_1, \ p_2, \ \neg p_3, \ p_4 \]
Basic Idea

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka "naming" atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

Assignment

\[ p_1, \ p_2, \ \neg p_3, \ p_4 \]

\[ x \geq 0, \ y = x + 1, \]
\[ \neg(y > 2), \ y < 1 \]
Basic Idea

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \quad p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

SAT Solver

Assignment

\[ p_1, \ p_2, \ \neg p_3, \ p_4 \quad x \geq 0, \ y = x + 1, \]
\[ \neg(y > 2), \ y < 1 \]

Unsatisfiable

\[ x \geq 0, \ y = x + 1, \ y < 1 \]

Theory Solver
**Basic Idea**

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]
\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

**SAT Solver**  
**Assignment**

\[ p_1, \ p_2, \ \neg p_3, \ p_4 \]

**x \geq 0, \ y = x + 1, \ \neg(y > 2), \ y < 1**

**New Lemma**

\[ \neg p_1 \lor \neg p_2 \lor \neg p_4 \]

**Unsatisfiable**

\[ x \geq 0, \ y = x + 1, \ y < 1 \]

**Theory Solver**
SAT + Theory solvers

New Lemma
\[ \neg p_1 \lor \neg p_2 \lor \neg p_4 \]

Unsatisfiable
\[ x \geq 0, \ y = x + 1, \ y < 1 \]

AKA Theory conflict

Theory Solver
procedure SmtSolver(F)
(F_p, M) := Abstract(F)

loop

(R, A) := SAT_solver(F_p)
if R = UNSAT then return UNSAT
S := Concretize(A, M)
(R, S’) := Theory_solver(S)
if R = SAT then return SAT
L := New_Lemma(S’, M)
Add L to F_p
Basic Idea

\( F: x \geq 0, y = x + 1, (y > 2 \lor y < 1) \)

Abstract (aka “naming” atoms)

\( F_p: p_1, p_2, (p_3 \lor p_4) \)

\( M: p_1 \equiv (x \geq 0), p_2 \equiv (y = x + 1), p_3 \equiv (y > 2), p_4 \equiv (y < 1) \)

A: Assignment
\( p_1, p_2, \neg p_3, p_4 \)

S: \( x \geq 0, y = x + 1, \neg(y > 2), y < 1 \)

L: New Lemma
\( \neg p_1 \lor \neg p_2 \lor \neg p_4 \)

S': Unsatisfiable
\( x \geq 0, y = x + 1, y < 1 \)

Theory Solver
**SAT + Theory solvers**

\[ F: x \geq 0, y = x + 1, (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ F_p : p_1, p_2, (p_3 \lor p_4) \]
\[ M: p_1 \equiv (x \geq 0), p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), p_4 \equiv (y < 1) \]

**SAT Solver**

\[ A: Assignment \]
\[ p_1, p_2, \neg p_3, p_4 \]

**Theory Solver**

\[ S: x \geq 0, y = x + 1, \neg (y > 2), y < 1 \]

**L: New Lemma**
\[ \neg p_1 \lor \neg p_2 \lor \neg p_4 \]

\[ S': Unsatisfiable \]
\[ x \geq 0, y = x + 1, y < 1 \]

**procedure** SMT_Solver(F)

(\( F_p, M \)) := Abstract(F)

loop

(R, A) := SAT_solver(\( F_p \))

if R = UNSAT then return UNSAT

S = Concretize(A, M)

(R, S’) := Theory_solver(S)

if R = SAT then return SAT

L := New_Lemma(S, M)

Add L to \( F_p \)

“Lazy translation” to DNF
State-of-the-art SMT solvers implement many improvements.
Incrementality
Send the literals to the Theory solver as they are assigned by the SAT solver

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1), \ p_5 \equiv (x < 2), \]
\[ p_1, \ p_2, \ p_4 \ | \ p_1, \ p_2, (p_3 \lor p_4), (p_5 \lor \neg p_4) \]

Partial assignment is already Theory inconsistent.
Efficient Backtracking

We don’t want to restart from scratch after each backtracking operation.
Efficient Lemma Generation (computing a small $S'$)

$$(R, S') := \text{Theory}\_\text{solver}(S)$$

When $R = \text{UNSAT}$ (i.e., $S$ is unsatisfiable),

$S' \subseteq S$ is also unsatisfiable

We say $S'$ is redundant

iff

Exists $S'' \subset S'$ which is also unsatisfiable.
Efficient Lemma Generation (computing a small $S'$)

Avoid lemmas containing redundant literals.

$p_1 \equiv (x \geq 0)$, $p_2 \equiv (y = x + 1)$,
$p_3 \equiv (y > 2)$, $p_4 \equiv (y < 1)$, $p_5 \equiv (x < 2)$,

$p_1, p_2, p_3, p_4 \mid p_1, p_2, (p_3 \lor p_4), (p_5 \lor \neg p_4)$

$\neg p_1 \lor \neg p_2 \lor \neg p_3 \lor \neg p_4$  

Imprecise Lemma
Theory Propagation
It is the SMT equivalent of unit propagation.

\[ p_1 \equiv (x \geq 0), \quad p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \quad p_4 \equiv (y < 1), \quad p_5 \equiv (x < 2), \]
\[ p_1, \ p_2 \ | \ p_1, \ p_2, \ (p_3 \lor p_4), \ (p_5 \lor \neg p_4) \]

\[ p_1, \ p_2 \] imply \( \neg p_4 \) by theory propagation

\[ p_1, \ p_2, \ \neg p_4 \ | \ p_1, \ p_2, \ (p_3 \lor p_4), \ (p_5 \lor \neg p_4) \]
Theory Propagation
It is the SMT equivalent of unit propagation.

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1), \ p_5 \equiv (x < 2), \]
\[ p_1, \ p_2 \models p_1, \ p_2, (p_3 \lor p_4), (p_5 \lor \neg p_4) \]
\[ p_1, \ p_2 \text{ imply } \neg p_4 \text{ by theory propagation} \]
\[ p_1, \ p_2, \neg p_4 \models p_1, \ p_2, (p_3 \lor p_4), (p_5 \lor \neg p_4) \]

Tradeoff between precision \( \times \) performance.
Deciding Equality + (uninterpreted) Functions

**Problem**: our procedure for Equality + UF does not support:
- Incrementality
- Efficient Backtracking
- Theory Propagation
- Lemma Learning
Incrementality (main problem):

We were processing the disequalities after we processed all equalities.

\[ p_1 \equiv a = b, \ p_2 \equiv b = c, \]
\[ p_3 \equiv d = e, \ p_4 \equiv a = c \]

\[ p_1, \neg p_4, p_2 \ \mid \ p_1, p_3 \lor \neg p_4, p_2 \lor p_4 \]

\[ a = b, \ a \neq c, \ b = c, \]
Incrementality (main problem):

We were processing the disequalities after we processed all equalities.

\[
p_1 \equiv a = b, \ p_2 \equiv b = c, \\
p_3 \equiv d = e, \ p_4 \equiv a = c
\]

\[
p_1, \neg p_4, p_2 \mid p_1, p_3 \lor \neg p_4, p_2 \lor p_4
\]

\[
a = b, \ a \neq c, \ b = c,
\]
Incrementality

Store the disequalities of a constant.

Very similar to the structure occurrences.

\[ a = b, a \neq c \]

\[ \text{diseqs}[b] = \{ a \neq c \} \]
\[ \text{diseqs}[c] = \{ a \neq c \} \]
Deciding Equality + (uninterpreted) Functions

**Incrementality**

Store the disequalities of a constant.

Very similar to the structure occurrences.

\[
a = b, \ a \neq c
\]

\[
diseqs[b] = \{ a \neq c \}
diseqs[c] = \{ a \neq c \}
\]

When we merge two equivalence classes, we must merge the sets diseqs. (circular lists again!)
Deciding Equality + (uninterpreted) Functions

Incrementality

Store the disequalities of a constant.

Very similar to the structure occurrences.

\[ a = b, \ a \neq c \]

\[ b \quad c \]

\[ a \]

\[ \text{diseqs}(b) = \{ a \neq c \} \]

\[ \text{diseqs}(c) = \{ a \neq c \} \]

When we merge two equivalence classes, we must merge the sets diseqs. (circular lists again!)

Before merging two equivalence classes, traverse one (the smallest) set of diseqs. (track the size of diseqs!)
Backtracking

Option 1: functional data-structures (too slow).

Option 2: trail stack (aka undo stack, fine grain backtracking)

Associate an undo operation to each update operation.

“Log” all update operations in a stack.

During backtracking execute the associated undo operations.
Backtracking

We can do better: coarse grain backtracking.

Minimize the size of the undo stack.

Do not track each small update, but a big operation (merge).
Deciding Equality + (uninterpreted) Functions

Backtracking

We can do better: coarse grain backtracking.
Minimize the size of the undo stack.
Do not track each small update, but a big operation (merge).

Let us change the union-find data-structure a little bit.

**Before:**
```
  s
 / 
|   |
b - r
|   |
|   |
a - c
```

**Fields:** find, size

**After:**
```
  s
/  \
|   |  <--- next element
|   |
|   |
a - b - c <--- r
```

**Fields:** root, next, size
Backtracking
We can do better: coarse grain backtracking.
Minimize the size of the undo stack.
Do not track each small update, but a big operation (merge).

Let us change the union-find data structure a little bit.

New design possibility:
We do not need to merge occurrences and diseqs.
We can access all occurrences and diseqs by traversing the next fields.

Before:
```
    s
   / 
  b   r
 /     
a      c
```

Fields: find, size

After:
```
    s
   / 
  n   
 /     
a ← b ← c ← r
```

Fields: root, next, size
Deciding Equality + (uninterpreted) Functions

New union-find:
Deciding Equality + (uninterpreted) Functions

New union-find:

What was updated?
root[s], root[r], next[b], next[s], size[b]
Deciding Equality + (uninterpreted) Functions

New union-find:

- What was updated?
  - root[c], root[r], next[b], next[s], size[b]

- We only need to store s in the undo stack!
Deciding Equality + (uninterpreted) Functions

What about the congruence table?

- hash table used to implement the congruence rule.

Let us use an additional field $cg$. It is only relevant for subterms: $v_3 \equiv f(a, v_1)$

Invariant: a constant (e.g., $v_3$) is in the table iff $cg[v_3] = v_3$

Otherwise, $cg[v_3]$ contains the subterm congruent to $v_3$

Example:

$v_3 \equiv f(a, v_1)$, $v_4 \equiv f(b, v_2)$

Assume $v_3$ and $v_4$ are congruent (i.e., $a = b$ and $v_1 = v_2$)

Moreover, $v_3$ is in the congruence table.

Then: $cg[v_4] = v_3$ and $cg[v_3] = v_3$
procedure Merge(a, b)
    a_r := root[a]; b_r := root[b]
    if a_r = b_r then return
    if not CheckDiseqs(a_r, b_r) then return
    if size[a] < size[b] then swap a, b; swap a_r, b_r
    AddToTrailStack(MERGE, b_r)
    RemoveParentsFromHashTable(b_r)
    c := b_r
    do
        root[c] := a_r
        c := next[c]
    while c ≠ b_r
    ReinsertParentsToHashTable(b_r)
    swap next[a_r], next[b_r]
    size[a_r] := size[a_r] + size[b_r]
Deciding Equality + (uninterpreted) Functions

**procedure** UndoMerge($b_r$)

$\quad a_r := \text{root}[b_r]$  
$\quad \text{size}[a_r] := \text{size}[a_r] – \text{size}[b_r]$  
$\quad \textbf{swap} \text{ next}[a_r], \text{next}[b_r]$  
$\quad \text{RemoveParentsFromHashTable}(b_r)$  
$\quad c := b_r$  
$\quad \textbf{do}$

$\quad \quad \text{root}[c] := b_r$  
$\quad \quad c := \text{next}[c]$  
$\quad \textbf{while} \ c \neq b_r$  
$\quad \textbf{for each} \ \text{parent} \ p \ \text{of} \ b_r$

$\quad \quad \textbf{if} \ p = \text{cg}[p] \ \textbf{or} \ \text{not congruent}(p, \ \text{cg}[p])$

$\quad \quad \quad \text{add} \ p \ \text{to hash table}$  
$\quad \quad \quad \text{cg}[p] := p$
Deciding Equality + (uninterpreted) Functions

```
procedure UndoMerge(b_r)
    a_r := root[b_r]
    size[a_r] := size[a_r] - size[b_r]
    swap next[a_r], next[b_r]
    RemoveParentsFromHashTable(b_r)
    c := b_r
do
    root[c] := b_r
    c := next[c]
while c ≠ b_r
    for each parent p of b_r
        if p = cg[p] or not congruent(p, cg[p])
            add p to hash table
            cg[p] := p
```

- p was in the hash table before and after the merge.
- p was in the hash table before but not after the merge.
Deciding Equality + (uninterpreted) Functions

Propagating equalities (and disequalities)

Store the atom occurrences of a constant.

\[ p_1 \equiv a = b, \ p_2 \equiv b = c, \]
\[ p_3 \equiv d = e, \ p_4 \equiv a = c \]

\[
\text{atom_occ}
\]
\[ a = \{ p_1, p_4 \} \]
\[ b = \{ p_1, p_2 \} \]
\[ c = \{ p_2, p_4 \} \]
\[ d = \{ p_3 \} \]
\[ e = \{ p_4 \} \]

When merging or adding new disequalities traverse these sets.
Propagating disequalities (hard case)

\[ v_1 \equiv f(a, b), \quad v_2 \equiv f(c, d) \]

Assume we know that

\[ v_1 \neq v_2 \]
\[ a = c \]

Then, \( b \neq d \)

More about that later.
Efficient Lemma Generation (computing a small $S'$)

In EUF (equality + UF) a minimal unsatisfiable set is composed on:
- $n$ equalities
- 1 disequality

It is easy to find the disequality $a \neq b$.

So, our problem consists in finding the minimal set of equalities that implies $a = b$. 
Efficient Lemma Generation (computing a small S’)

First idea:
If \( a = b \) is implied by a set of equalities, then \( a \) and \( b \) are in the same equivalence class.

Store all equalities used to “create” the equivalence class.

\[
p_1 \equiv (a = c), \quad p_2 \equiv (b = c),
\]
\[
p_3 \equiv (s = r), \quad p_4 \equiv (c = r)
\]
\[
p_1, p_2, p_3, p_4, \ldots | \ldots
\]

Too imprecise for justifying \( a = b \).
We need only \( p_1, p_2 \).

The equivalence class was “created” using \( p_1, p_2, p_3, p_4 \).
Deciding Equality + (uninterpreted) Functions

Efficient Lemma Generation (computing a small $S'$)

Second idea: Store a “proof tree”.
Each constant $c$ has a non-redundant “proof” for $c = \text{root}[c]$.
The proof is a path from $c$ to $\text{root}[c]$

$$p_1 \equiv (a = c), \ p_2 \equiv (b = c), \ p_3 \equiv (s = r), \ p_4 \equiv (c = r)$$
procedure Merge(a, b, p_i)
a_r := root[a]; b_r := root[b]
if a_r = b_r then return
if not CheckDiseqs(a_r, b_r) then return
if size[a] < size[b] then swap a, b; swap a_r, b_r
InvertPathFrom(b, b_r); AddProofEdge(b, a, p_i)
AddToTrailStack(MERGE, b_r, b)
...

Deciding Equality + (uninterpreted) Functions
Deciding Equality + (uninterpreted) Functions

Non redundant proof for $a = b$

$p_1, ..., p_n, q_1, ..., q_m$

Common ancestor in the proof tree.
Extract a non-redundant proof for \(a = r\), \(a = b\) and \(a = s\).
What about congruence?

New form of justification for an edge in the “proof tree”.

\[ v_1 \equiv f(b), \ v_2 \equiv f(c) \]
Deciding Equality + (uninterpreted) Functions

What about congruence?

New form of justification for an edge in the “proof tree”.

\[ v_1 \equiv f(b), \; v_2 \equiv f(c) \]

When computing the “proof” for \( a = v_2 \)

Recursive call for computing the proof for \( v_1 = v_2 \)

Result: \( \{p_1, p_2\} \)
The new algorithm may compute redundant proofs for EUF.

Using notation $a = b$ for $p \equiv a = b$, and $p$ assigned by SAT solver

\[
\begin{align*}
    f_1(a_1) &= a_1 = a_2 = f_1(a_5) \\
    f_2(a_1) &= a_2 = a_3 = f_2(a_5) \\
    f_3(a_1) &= a_3 = a_4 = f_3(a_5) \\
    f_4(a_1) &= a_4 = a_5 = f_4(a_5)
\end{align*}
\]
The new algorithm may compute redundant proofs for EUF.

Using notation $a = b$ for $p \equiv a = b$, and $p$ assigned by SAT solver

\[
\begin{align*}
  f_1(a_1) &= a_1 = a_2 = f_1(a_5) \\
  f_2(a_1) &= a_2 = a_3 = f_2(a_5) \\
  f_3(a_1) &= a_3 = a_4 = f_3(a_5) \\
  f_4(a_1) &= a_4 = a_5 = f_4(a_5)
\end{align*}
\]

Two non redundant proofs $f_2(a_1) = f_2(a_5)$:

\[
\begin{align*}
  \{p_2, q_2, s_2\} & \text{ using transitivity} \\
  \{q_1, q_2, q_3, q_4\} & \text{ using congruence } a_1 = a_5
\end{align*}
\]

Similar for $f_1, f_3, f_4$. 
The new algorithm may compute redundant proofs for EUF.

Using notation $a \equiv b$ for $p \equiv a = b$, and $p$ assigned by SAT solver

\[
f_1(a_1) = \begin{cases} p_1 \quad & q_1 \quad & s_1 \quad & f_1(a_5) \\ 0x1\end{cases}
\]
\[
f_2(a_1) = \begin{cases} p_2 \quad & q_2 \quad & s_2 \quad & f_2(a_5) \\ 0x1\end{cases}
\]
\[
f_3(a_1) = \begin{cases} p_3 \quad & q_3 \quad & s_3 \quad & f_3(a_5) \\ 0x1\end{cases}
\]
\[
f_4(a_1) = \begin{cases} p_4 \quad & q_4 \quad & s_4 \quad & f_4(a_5) \\ 0x1\end{cases}
\]

Two non redundant proofs $f_2(a_1) = f_2(a_5)$:
\[
\{p_2, q_2, s_2\} \text{ using transitivity}
\]
\[
\{q_1, q_2, q_3, q_4\} \text{ using congruence } a_1 = a_5
\]

Similar for $f_1, f_3, f_4$.

So there are 16 proofs for
\[
g(f_1(a_1), f_2(a_1), f_3(a_1), f_4(a_1)) = g(f_1(a_5), f_2(a_5), f_3(a_5), f_4(a_5))
\]

The only non redundant is $\{q_1, q_2, q_3, q_4\}$
Some benchmarks are very hard for our procedure.

\[ p_1 \lor a_1 = c_0, \neg p_1 \lor a_1 = c_1, \quad p_1 \lor b_1 = c_0, \neg p_1 \lor b_1 = c_1, \]
\[ p_2 \lor a_2 = c_0, \neg p_2 \lor a_2 = c_1, \quad p_2 \lor b_2 = c_0, \neg p_2 \lor b_2 = c_1, \]
\[ \ldots, \]
\[ p_n \lor a_n = c_0, \neg p_n \lor a_n = c_1, \quad p_n \lor b_n = c_0, \neg p_n \lor b_n = c_1, \]
\[ f(a_n, \ldots, f(a_2, a_1)\ldots) \neq f(b_n, \ldots, f(b_2, b_1)\ldots) \]
Some benchmarks are very hard for our procedure. 

\[ p_1 \lor a_1 = c_0, \neg p_1 \lor a_1 = c_1, \quad p_1 \lor b_1 = c_0, \neg p_1 \lor b_1 = c_1, \]
\[ p_2 \lor a_2 = c_0, \neg p_2 \lor a_2 = c_1, \quad p_2 \lor b_2 = c_0, \neg p_2 \lor b_2 = c_1, \]
\[ \ldots, \]
\[ p_n \lor a_n = c_0, \neg p_n \lor a_n = c_1, \quad p_n \lor b_n = c_0, \neg p_n \lor b_n = c_1, \]
\[ f(a_n, \ldots, f(a_2, a_1)\ldots) \neq f(b_n, \ldots, f(b_2, b_1)\ldots) \]

Lemmas learned during the search are not useful. They only use atoms that are already in the problem!
Some benchmarks are very hard for our procedure.

\[ p_1 \lor a_1 = c_0, \neg p_1 \lor a_1 = c_1, \quad p_1 \lor b_1 = c_0, \neg p_1 \lor b_1 = c_1, \]
\[ p_2 \lor a_2 = c_0, \neg p_2 \lor a_2 = c_1, \quad p_2 \lor b_2 = c_0, \neg p_2 \lor b_2 = c_1, \]
\[ \ldots, \]
\[ p_n \lor a_n = c_0, \neg p_n \lor a_n = c_1, \quad p_n \lor b_n = c_0, \neg p_n \lor b_n = c_1, \]
\[ f(a_n, \ldots, f(a_2, a_1)\ldots) \neq f(b_n, \ldots, f(b_2, b_1)\ldots) \]

Lemmas learned during the search are not useful.
They only use atoms that are already in the problem!
Solution: congruence rule suggests which new atoms must be created.
Deciding Equality + (uninterpreted) Functions

Some benchmarks are very hard for our procedure.

\[ p_1 \lor a_1 = c_0, \neg p_1 \lor a_1 = c_1, \quad p_1 \lor b_1 = c_0, \neg p_1 \lor b_1 = c_1, \]
\[ p_2 \lor a_2 = c_0, \neg p_2 \lor a_2 = c_1, \quad p_2 \lor b_2 = c_0, \neg p_2 \lor b_2 = c_1, \]
\[ \ldots, \]
\[ p_n \lor a_n = c_0, \neg p_n \lor a_n = c_1, \quad p_n \lor b_n = c_0, \neg p_n \lor b_n = c_1, \]
\[ f(a_n, \ldots, f(a_2, a_1)\ldots) \neq f(b_n, \ldots, f(b_2, b_1)\ldots) \]

Solution: congruence rule suggests which new atoms must be created.

Whenever, the congruence rules
\[ a_i = b_i, \quad a_j = b_j \implies f(a_i, a_j) = f(b_i, b_j) \]
is used to (immediately) deduce a conflict. Add the clause:
\[ a_i \neq b_i \lor a_j \neq b_j \lor f(a_i, a_j) = f(b_i, b_j) \]
Solution: congruence rule suggests which new atoms must be created.

Whenever, the congruence rules

\[ a_i = b_i, \ a_j = b_j \implies f(a_i, a_j) = f(b_i, b_j) \]

is used to (immediately) deduce a conflict. Add the clause:

\[ a_i \neq b_i \lor a_j \neq b_j \lor f(a_i, a_j) = f(b_i, b_j) \]

“Dynamic Ackermannization”

It allows the solver to perform the missing disequality propagation.
We can solve the QF_UF SMT-Lib benchmarks!