CS257: Introduction to Automated Reasoning

Theory Solvers

Stanford University

CENTAUR

Roadmap for Today

Theory Solvers

- Difference Logic
- Equality and Uninterpreted Functions
- Arrays
- Strings

Theory Solvers

A **theory solver** for T is a decision procedure for determining whether a conjunction of literals is T-satisfiable

Theory solvers are crucial building blocks in modern SMT solvers

A Fragment of Arithmetic: Difference Logic

In **difference logic**, we are interested in the satisfiability of a conjunction of arithmetic atoms.

Each atom is of the form $x - y \bowtie c$, where x and y are variables, c is a numeric constant, and $\bowtie \in \{=, <, \leq, >, \geq\}$.

The variables can range over either the **integers** (QF_IDL) or the **reals** (QF_RDL).

Difference Logic

The first step is to rewrite everything in terms of $\leq$:

- $x - y = c \implies x - y \leq c \land x - y \geq c$
- $x - y \geq c \implies y - x \leq -c$
- $x - y > c \implies y - x < -c$
- $x - y < c \implies x - y \leq c - 1$ (integers)
- $x - y < c \implies x - y \leq c - \delta$ (reals)

Note: using δ requires some additional infrastructure which we will not cover here

Difference Logic

Now we have a conjunction of literals, all of the form $x - y \leq c$.

From these literals, we form a weighted directed graph with a vertex for each variable.

For each literal $x - y \leq c$, there is an edge $x \xrightarrow{c} y$.

The set of literals is satisfiable iff there is no cycle for which the sum of the weights on the edges is negative.

There are a number of efficient algorithms for detecting negative cycles in graphs

- e.g., Bellman-Ford, $O(|V| \cdot |E|)$

Difference Logic Example

$$x - y = 5 \ \wedge \ z - y \geq 2 \ \wedge \ z - x > 2 \ \wedge \ w - x = 2 \ \wedge \ z - w < 0$$

$$\begin{aligned}
x - y &= 5 \\
z - y &\geq 2 \\
z - x &> 2 \\
w - x &= 2 \\
z - w &< 0
\end{aligned}$$

Difference Logic Example

$$x - y = 5 \;\wedge\; z - y \geq 2 \;\wedge\; z - x > 2 \;\wedge\; w - x = 2 \;\wedge\; z - w < 0$$

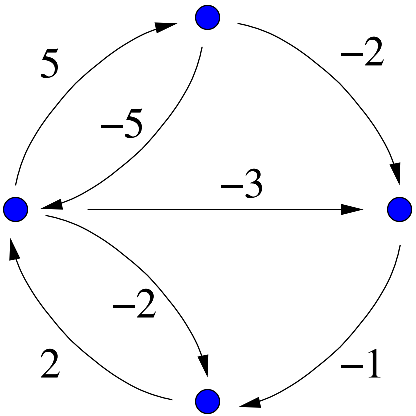
$$\begin{aligned} x - y &= 5 \\ z - y &\geq 2 \\ z - x &> 2 \qquad \Rightarrow \\ w - x &= 2 \\ z - w &< 0 \end{aligned}$$

Difference Logic Example

$$x - y = 5 \ \wedge \ z - y \geq 2 \ \wedge \ z - x > 2 \ \wedge \ w - x = 2 \ \wedge \ z - w < 0$$

$$\begin{array}{lcl} x - y = 5 & & x - y \leq 5 \wedge y - x \leq -5 \\ z - y \geq 2 & & y - z \leq -2 \\ z - x > 2 & \Rightarrow & x - z \leq -3 \\ w - x = 2 & & w - x \leq 2 \wedge x - w \leq -2 \\ z - w < 0 & & z - w \leq -1 \end{array}$$

Difference Logic Example



Theory Solvers as Satisfiability Proof Systems

How do we determine whether a set of literals is T-satisfiable?

For many theories, we can use the framework of **satisfiability proof systems**.

Notation and Assumptions

A literal is **flat** if it is of the form: $x = y$, $x \neq y$, or $x = f(\vec{z})$, where x, y are variables, f is a function symbol and $\vec{z}$ is a tuple of 0 or more variables.

Any set of literals can be converted to an equisatisfiable flat set of literals by introducing new variables.

Example

$x + y > 0 = \mathtt{T},\ \ y = f(g(z))$

$\Rightarrow$

$v_1 = x + y, v_2 = 0, v_3 = \mathtt{T}, v_3 = v_1 > v_2, v_4 = g(z), y = f(v_4)$

For the proof systems we present next, we assume that all literals are flat

Notation and Assumptions

For tuples $\vec{v}$ and $\vec{w}$ of the same size, we write $\vec{v} = \vec{w}$ as an abbreviation for the set of pairwise equalities between corresponding elements of the tuples

Proof states (besides $\textsc{sat}$, $\textsc{unsat}$) are sets of formulas, and the satisfiable states are those that are T-satisfiable (plus $\textsc{sat}$)

We use Γ to refer to the current proof state in rule premises

When presenting rules, a list formulas as a premise means these are all required to be in the starting proof state (i.e., we omit $\in \Gamma$, leaving it implicit)

We list in the conclusion of each rule only the literals to be **added** to the proof state and assume no literals are ever deleted

From now on, we also assume that if applying a rule does not change Γ, then that rule is **not applicable** to Γ, i.e., Γ is irreducible with respect to that rule

A Satisfiability Proof System for QF_UF

As an example, we present a simple satisfiability proof system R_{UF} for QF_UF

contr	$\dfrac{x = y, x \neq y}{\text{UNSAT}}$	refl	$\dfrac{x \text{ occurs in } \Gamma}{x = x}$
symm	$\dfrac{x = y}{y = x}$	trans	$\dfrac{x = y, y = z}{x = z}$
cong	$\dfrac{x = f(\vec{v}), y = f(\vec{w}), \vec{v} = \vec{w}}{x = y}$	sat	$\dfrac{\text{no other rule applies}}{\text{SAT}}$

Is R_{UF} **sound**? **terminating**?

Soundness, Termination, Completeness

By inspecting each rule, all but sat are clearly satisfiability preserving, so it follows that R_{UF} is **refutation sound**

Since R_{UF} only introduces equalities between variables and never introduces new variables (and there are only a finite number of possible equalities between existing variables), every strategy for R_{UF} must **terminate**

Solution soundness can be shown by constructing an interpretation from any proof state to which sat applies and is the most challenging step

Theorem If sat applies to Γ, then Γ is satisfiable

Proof Sketch Let $x \sim t$ iff $x = t \in \Gamma$. We can show that $\sim$ is an equivalence relation. Let the domain of I be the equivalence classes of $\sim$. Let $\alpha = [v]_\sim$ for some arbitrary variable $v \in \Gamma$. Let $x^I = \text{ if } x \in \Gamma \text{ then } [x]_\sim \text{ else } \alpha$. For a unary function symbol f, let $f^I = \lambda e.\ \text{if } f(x) \text{ occurs in } \Gamma \text{ for some } x \in e, \text{ then } [f(x)]_\sim \text{ else } \alpha$. Define f^I for non-unary f similarly. We can show that $I \models \Gamma$.

Theory of Arrays T_A

Signature:

- Equality: Yes
- $\Sigma^S = \{A, I, E\}$ (for arrays, indices, elements)
- $\Sigma^F = \{read, write\}$
- $\mathit{sort}(read) = \langle A, I, E \rangle$, $\mathit{sort}(write) = \langle A, I, E, A \rangle$

Useful for modeling memories or array data structures.

Example

```
void ReadBlock(int data[], int x, int len)
{
  int i = 0;
  int next = data[0];
  for (; i < next && i < len; i = i + 1) {
    if (data[i] == x)
      break;
    else
      Process(data[i]);
  }
  assert(i < len);
}
```

One path through this code can be translated using the theory of arrays as:

$$i = 0 \wedge next = read(data, 0) \wedge i < next \wedge i < len \wedge read(data, i) = x \wedge \neg(i < len)$$

Semantics of T_A

Recall that a theory is made up of a signature and a class of structures.

How do we define which structures are allowed?

One way is to say it is all structures satisfying a set of sentences. These sentences are the **axioms** of the theory.

Not all theories can be finitely axiomatized, but the theory of arrays can. It requires only three axioms:

$$\forall\, a{:}A.\ \forall\, i{:}I.\ \forall\, v{:}E.\ \ read(write(a,i,v),i) = v \ , \tag{RW1}$$

$$\forall\, a{:}A.\ \forall\, i,j{:}I.\ \forall\, v{:}E.\ \ i \neq j \rightarrow read(write(a,i,v),j) = read(a,j) \ , \tag{RW2}$$

$$\forall\, a,b{:}A.\ \ (\forall\, i{:}I.\, read(a,i) = read(b,i)) \rightarrow a = b \ . \tag{EX}$$

A Satisfiability Proof System for T_A

The satisfiability proof system R_A for T_A extends the proof system for QF_UF with the following rules:

RIntro1 $$\frac{a = write(b, i, v)}{v = read(a, i)}$$

RIntro2 $$\frac{a = write(b, i, v), x = read(c, j) \qquad a = c \in \Gamma \text{ or } b = c \in \Gamma}{i = j \qquad\qquad read(a, j) = read(b, j)}$$

Ext $$\frac{a \neq b}{read(a, k_{a,b}) \neq read(b, k_{a,b})}$$

where for each pair (a, b) of array variables, $k_{a,b}$ denotes a distinguished fresh variable of sort I.

Example

Let $\Gamma = write(a, i, read(b, i)) = write(b, i, read(a, i)), a \neq b$

Example

Let $\Gamma = write(a, i, read(b, i)) = write(b, i, read(a, i)), a \neq b$

$$\cfrac{a' = write(a, i, v), b' = write(b, i, w), v = read(b, i), w = read(a, i), a' = b', a \neq b}{\cfrac{x = read(a, k), y = read(b, k), x \neq y}{\cfrac{i = k}{\cfrac{x = w, v = y}{\cfrac{v = read(a', i)}{\cfrac{w = read(b', i)}{\cfrac{w = v}{\cfrac{x = y}{\text{UNSAT}}}}}}} \quad \cfrac{x' = read(a', k), x = x'}{\cfrac{i = k}{\ldots} \quad \cfrac{y' = read(y', k), y' = y}{\cfrac{x' = y'}{\cfrac{x = y}{\text{UNSAT}}}}}}}$$

A Satisfiability Proof System for T_A

The satisfiability proof system R_A for T_A extends the proof system for QF_UF with the following rules:

RIntro1 $$\frac{a = write(b, i, v)}{v = read(a, i)}$$

RIntro2 $$\frac{a = write(b, i, v), x = read(c, j) \qquad a = c \in \Gamma \text{ or } b = c \in \Gamma}{i = j \qquad read(a, j) = read(b, j)}$$

Ext $$\frac{a \neq b}{read(a, k_{a,b}) \neq read(b, k_{a,b})}$$

where for each pair (a, b) of array variables, $k_{a,b}$ denotes a distinguished fresh variable of sort I

Is R_A sound? terminating?

Soundness, Termination, and Completeness

Refutation soundness is straightforward and follows from the T_A axioms.

Termination follows from the following argument. Once we add all of the $k_{a,b}$ variables, no rule introduces new variables. There are only a finite number of terms that match the conclusions that can be constructed with a finite number of variables, so eventually, Γ will become reducible only by the sat rule.

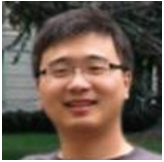
Solution soundness is again by constructing an interpretation but is much more involved. Essentially, we construct an interpretation much as we did for R_{UF}, but then we modify it to ensure the array axioms are satisfied.

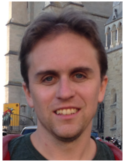
More details in Section 5 of Jovanović and Barrett, "Being Careful about Theory Combination", 2013.

Reasoning about Strings

Joint work with David Brumley, Morgan Deters, Tianyi Liang, Andres Nötzli, Andrew Reynolds, Cesare Tinelli, Nestan Tsiskaridze, and Maverick Woo



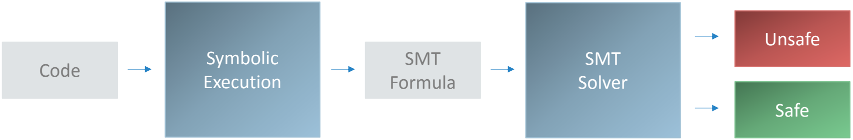





Motivation: Symbolic Execution

Symbolic Execution

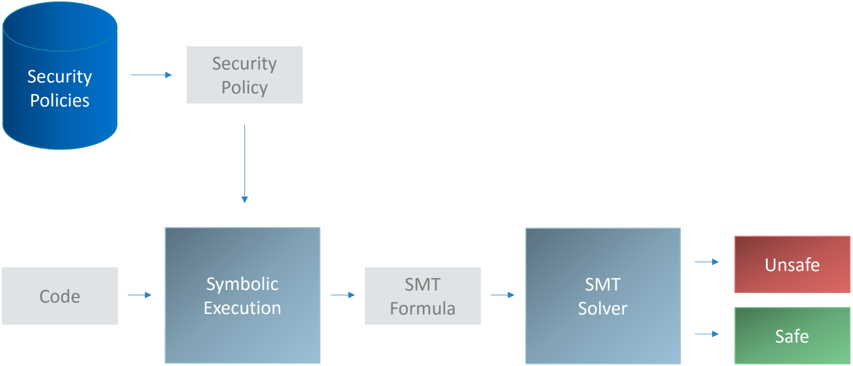
- **Enumerate program paths** that end in a bad state
  - (e.g., invalid memory access)
- Represent program **inputs** as SMT **variables**
- Translate **statements** in the path into **constraints** on the variables
- Constraints represent **all possible executions** along the path
- Solving the constraints determines whether the path is **feasible**



Example: Symbolic Execution for Security

Security Vulnerabilities

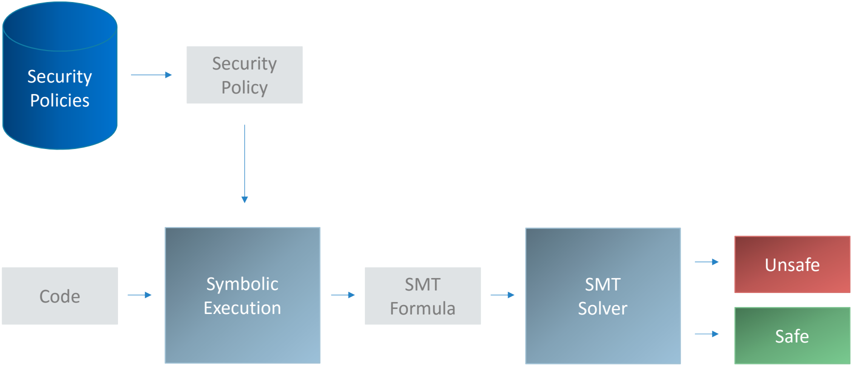
- Input: **code** and **security policy**
- **Symbolic execution**: generates formula satisfiable iff code can violate security policy
- **SMT solver**: returns a solution or proves that none exists



String Analysis

Strings in Symbolic Execution

- Input code may manipulate **strings**



Basic Theory of Strings

Alphabet

A fixed **finite** set of characters

Constants

Empty string $\epsilon : String$

Character string $c : String$ for all $c \in A$

Operators

Length $|\_| : String \rightarrow Int$

Concatenation $\_ \mathbin{+\!\!+} \_ : String \times String \rightarrow String$

Equality $\_ = \_ : String \times String \rightarrow Bool$

Membership $\_ \in \_ : String \times RegEx$

A Theory Solver for Strings

Alphabet

A fixed **finite** set o

Constants

Empty string $\epsilon : St$

Character string $c : St$

Operators

Length $|\_| : St$

Concatenation $\_ +\!\!+ \_ :$

Equality $\_ = \_ : String \times String \to Bool$

Membership $\_ \in \_ : String \times RegEx$

Challenge: complexity

concatenation + equality: **word equations problem**

- Decidable in PSPACE

+ length

- Decidability open

+ replace (all instances of some substring)

- Undecidable

A Theory Solver for Strings

Alphabet

A fixed **finite** set o

Constants

Empty string $\epsilon : St$

Character string $c : St$

> **Pragmatic approach**
>
> - Rule-based proof system
> - Use existing arithmetic theory solver
> - Embrace incompleteness

Operators

Length	$\lvert\_\rvert : String \to Int$
Concatenation	$\_ \mathbin{+\!\!+} \_ : String \times String \to String$
Equality	$\_ = \_ : String \times String \to Bool$
Membership	$\_ \in \_ : String \times RegEx$

Satisfiability Proof System for Strings

Proof States

A **proof state** is either:

- One of the distinguished states $\textsc{sat}$, $\textsc{unsat}$
- A pair (S, A), where S contains **string** constraints and A contains **arithmetic** constraints

Assumptions

- All literals are flat
- For every string variable x, there exists a variable ℓ_x, such that $\ell_x = |x| \in S$
- Ignore regular expression membership for now

Notation

Definitions

- $T(S)$ denotes all terms in S
- $S \vDash \phi$ means that ϕ follows from S using the rules of QF_UF
- $A \vDash_{LIA} \phi$ means that ϕ follows from A in the theory of linear integer arithmetic

Rewrite rules for string length

- $|\epsilon| \downarrow = 0$
- $|c| \downarrow = 1$, where $c \in A$
- $|s_1 +\!\!+ \cdots +\!\!+ s_n| \downarrow = |s_1| + \cdots |s_n|$

Core Rules

A-Conf $\dfrac{A \vDash_{LIA} \bot}{\text{UNSAT}}$ A-Prop $\dfrac{A \vDash_{LIA} s = t \quad s, t \in T(S)}{S := S, s = t}$

S-Conf $\dfrac{S \vDash \bot}{\text{UNSAT}}$ S-Prop $\dfrac{S \vDash s = t \quad s, t \in T(S) \quad s, t \text{ are } \Sigma_{LIA}\text{-terms}}{A := A, s = t}$

S-A $\dfrac{x, y \in T(S) \cap T(A) \quad x, y : Int}{A := A, x = y \quad A := A, x \neq y}$

L-Intro $\dfrac{s \in T(S) \quad s : String}{S := S, |s| = |s| \downarrow}$ L-Valid $\dfrac{x \in T(S) \quad x : String}{S := S, x = \epsilon \quad A := A, \ell_x > 0}$

Const-Conf $\dfrac{S \vDash c = d \quad c, d \in A, c \neq d}{\text{UNSAT}}$ Sat $\dfrac{\text{no other rule applies}}{\text{SAT}}$

Example

Let $S_0 = \{x = y +\!\!+\, x +\!\!+\, z, y = \text{“}a\text{”}, \ell_x = |x|, \ell_y = |y|, \ell_z = |z|\}, A_0 = \varnothing$

Example

Let $S_0 = \{x = y \mathbin{+\!\!+} x \mathbin{+\!\!+} z, y = \text{“}a\text{”}, \ell_x = |x|, \ell_y = |y|, \ell_z = |z|\}, A_0 = \varnothing$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(|y \mathbin{+\!\!+} x \mathbin{+\!\!+} z| = |y| + |x| + |z|, \varnothing)}}{(|\text{“}a\text{”}| = 1, \varnothing)}}{(\varnothing, \ell_x = \ell_y + \ell_x + \ell_z)}}{(\varnothing, \ell_y = 1)}}{\dfrac{\dfrac{\dfrac{(z = \epsilon, \varnothing)}{(|\epsilon| = 0, \varnothing)}}{(\varnothing, \ell_z = 0)}}{\text{UNSAT}} \qquad \dfrac{(\varnothing, \ell_z > 0)}{\text{UNSAT}}}$$

Concatenation Rules

Given a variable x, we can recursively expand x by substituting using equalities from S whose right-hand sides are concatenation terms until this is no longer possible

If t is the result, we say that $S \vDash^*_{\mathbin{+\!\!+}} x = t$

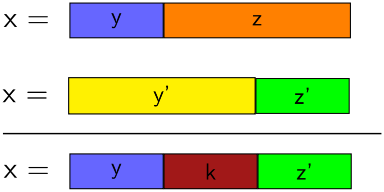
We write $\vec{z}$ as a short-hand for a concatenation of one or more variables

$$\text{C-Eq} \quad \frac{S \vDash^*_{\mathbin{+\!\!+}} x = \vec{z} \qquad S \vDash^*_{\mathbin{+\!\!+}} y = \vec{z}}{S := S, x = y}$$

$$\text{C-Split} \quad \frac{S \vDash^*_{\mathbin{+\!\!+}} x = \vec{w} \mathbin{+\!\!+} y \mathbin{+\!\!+} \vec{z} \qquad S \vDash^*_{\mathbin{+\!\!+}} x = \vec{w} \mathbin{+\!\!+} y' \mathbin{+\!\!+} \vec{z}'}{\begin{array}{c} A := A, \ell_y > \ell_{y'}; S := S, y = y' \mathbin{+\!\!+} k \\ A := A, \ell_y < \ell_{y'}; S := S, y' = y \mathbin{+\!\!+} k \\ A := A, \ell_y = \ell_{y'}; S := S, y = y' \end{array}}$$

Example of C-Split

C-Split $$\dfrac{S \models^*_{++} x = \vec{w} ++ y ++ \vec{z} \qquad S \models^*_{++} x = \vec{w} ++ y' ++ \vec{z}'}{\begin{array}{c} A := A, \ell_y > \ell_{y'}; S := S, y = y' ++ k \\ A := A, \ell_y < \ell_{y'}; S := S, y' = y ++ k \\ A := A, \ell_y = \ell_{y'}; S := S, y = y' \end{array}}$$



Properties of the proof system

Is the proof system sound? terminating?

Properties of the proof system

Is the proof system sound? terminating?

The proof system is **refutation sound**. This can easily be checked by examining each rule.

The proof system is **not terminating**. For pathological cases, C-Split can be applied infinitely often

Since it is not terminating, it is also **not complete**

However, it is **solution sound**. Proving this is highly non-trivial

Iterating to Improve the Solver

The first version of the proof system was implemented in 2014

Based on requests and feedback from users, a number of iterative improvements have been made

More String Operators

SMT user: That's great but I need more operators!

Iterate and Improve

- Extend the theory by adding **new operators**
- Implement by **reducing** to the core theory
- $substr(x, n, m)$: the maximal **substring** of x, starting at position n, with length at most m
- $contains(x, y)$: true iff x **contains** y as a substring
- $index\_of(x, y, n)$: position of **the first occurrence** of y in x, starting from position n
- $replace(x, y, z)$: the result of **replacing the first occurrence** of x in y by z

More String Operators

$$
\begin{array}{rcl}
[\![x = substr(y,n,m)]\!] & = & \texttt{ite}(\;\; 0 \le n < |y| \land 0 < m, \\
 & & \qquad\quad y = z_1 \mathbin{+\!\!+} x \mathbin{+\!\!+} z_2 \land |z_1| = n \land |z_2| = |y| \mathbin{\dot{-}} (m+n), \\
 & & \qquad\quad x = \epsilon \;) \\
[\![contains(y,z)]\!] & = & \exists\, k.\, 0 \le k \le |y| - |z| \land substr(y,k,|z|) = z \\
[\![x = index\_of(y,z,n)]\!] & = & \texttt{ite}(\;\; 0 \le n \le |y| \land contains(y',z),\; substr(y',x',|z|) = z \land \\
 & & \qquad\quad \neg contains(substr(y',0,x'+|z|-1),z),\; x = -1 \;) \\
 & & \text{with } y' = substr(y,n,|y|-n) \text{ and } x' = x - n \\
[\![x = replace(y,z,w)]\!] & = & \texttt{ite}(\;\; contains(y,z) \land z \ne \epsilon,\; x = z_1 \mathbin{+\!\!+} w \mathbin{+\!\!+} z_2 \land \\
 & & \qquad\quad y = z_1 \mathbin{+\!\!+} z \mathbin{+\!\!+} z_2 \land index\_of(y,z,0) = |z_1|,\; x = y \;)
\end{array}
$$

Note: $x \mathbin{\dot{-}} y = \max(x-y, 0)$

Reasoning about High-Level Operators

SMT user: That's great but now it's too slow!

Iterate and Improve

- Extend the implementation to **reason directly** on the new operators
- **How?**
  - Keep formulas with **original operators**
  - Periodically try to **simplify** them based on new knowledge

Reasoning about High-Level Operators

Examples using $contains$

$$
\begin{array}{rcll}
contains(l_1, l_2) & \rightarrow & \top & \text{if } l_1 \text{ contains } l_2 \\
contains(l_1, l_2) & \rightarrow & \bot & \text{if } l_1 \text{ does not contain } l_2 \\
contains(l_1, l_2 +\!\!+ \vec{t}) & \rightarrow & \bot & \text{if } l_1 \text{ does not contain } l_2 \\
contains(l_1, l_2 +\!\!+ \vec{t}) & \rightarrow & \bot & \text{if } contains(l_1 \setminus l_2, \vec{t}) \rightarrow^* \bot \\
contains(l_1, x +\!\!+ \vec{t}) & \rightarrow & \bot & \text{if } contains(l_1, \vec{t}) \rightarrow^* \bot \\
contains(l_1 +\!\!+ \vec{t}, l_2) & \rightarrow & \top & \text{if } l_1 \text{ contains } l_2 \\
contains(x +\!\!+ \vec{t}, s) & \rightarrow & \top & \text{if } contains(\vec{t}, s) \rightarrow^* \top \\
contains(t +\!\!+ \vec{s}, t +\!\!+ \vec{u}) & \rightarrow & \top & \text{if } contains(\vec{s}, \vec{u}) \rightarrow^* \top \\
contains(l_1 +\!\!+ \vec{t}, l_2) & \rightarrow & contains(\vec{t}, l_2) & \text{if } l_1 \sqcup_l l_2 = \epsilon \\
contains(\vec{t} +\!\!+ l_1, l_2) & \rightarrow & contains(\vec{t}, l_2) & \text{if } l_1 \sqcup_r l_2 = \epsilon
\end{array}
$$

$$
\begin{array}{rcll}
contains(\epsilon, t) = \top & \rightarrow & \epsilon = t & \\
contains(\vec{t}_1 +\!\!+ l_1 +\!\!+ \vec{t}_2, l_2) = \top & \rightarrow & \vee_{i=1}^{2} contains(\vec{t}_i, l_2) = \top & \text{if } l_1 \sqcup_r l_2 = l_1 \sqcup_l l_2 = \epsilon
\end{array}
$$

Reasoning about High-Level Operators

SMT user: That's great but I have a few really hard problems!

Iterate and Improve

- **Supercharge** the simplifier
- Many simplifications are **conditional**
- Build a **mini-inference engine** inside the simplifier to derive more conditions

Reasoning about High-Level Operators

Examples of Conditional Simplifications based on String Length

$$
\begin{array}{rcll}
t = s & \to & \bot & \text{if} \quad \vdash |t| \geq |s| + 1 \\
t = s \mathbin{+\!\!+} r \mathbin{+\!\!+} q & \to & t = s \mathbin{+\!\!+} q \wedge r = \epsilon & \text{if} \quad \vdash |s| + |q| \geq |t| \\
contains(t, s) & \to & t = s & \text{if} \quad \vdash |s| \geq |t| \\
substr(t, v, w) & \to & \epsilon & \text{if} \quad \vdash 0 > v \vee v \geq |t| \vee 0 \geq w \\
substr(t \mathbin{+\!\!+} s, v, w) & \to & substr(s, v - |t|, w) & \text{if} \quad \vdash v \geq |t| \\
substr(s \mathbin{+\!\!+} t, v, w) & \to & substr(s, v, w) & \text{if} \quad \vdash |s| \geq v + w \\
substr(t \mathbin{+\!\!+} s, 0, w) & \to & t \mathbin{+\!\!+} substr(s, 0, w - |t|) & \text{if} \quad \vdash w \geq |t| \\
index\_of(t, s, v) & \to & \texttt{ite}(substr(t, v) = s, v, -1) & \text{if} \quad \vdash v + |s| \geq |t|
\end{array}
$$

Too Domain-Specific?

SMT user: Wow! - but after all that, I bet you really overfit to that one symbolic execution domain, right?

Amazon Automated Reasoning Group:

- Hey! - we really like your string solver...
- ...and we are calling it a few **billion** times a day...
- to secure access control policies in the cloud for our customers!

Zelkova

Security Policy

```
(allow,
 principal    : *,
 action       : getObject,
 resource     : cs240/*,
 condition    : (StringEquals, aws:sourceVpc, vpc-111bbb222),
                (StringLike, s3:prefix, cs240/Exam*))
```

SMT Encoding

a = "getObject" ∧ r = "cs240/*" ∧ vpcExists ∧ vpc = "vpc-111bbb222" ∧ s3PrefixExists ∧ "cs240/Exam" prefixOf s3Prefix

SMT Solvers (cvc5 and z3)

Strings and RegExp	Bitvectors	Arithmetic

One More Thing

Amazon Automated Reasoning Group:

- Just one small thing though...
- We use a lot of **regular expressions**
- I don't suppose you could speed those up a bit?

Reasoning about Regular Expressions

Regular Expression Example

$$x \in [0..9]^* +\!\!+ \texttt{"a"} +\!\!+ \Sigma^* +\!\!+ \texttt{"b"} +\!\!+ \Sigma^*$$
$$x \notin [0..9]^* +\!\!+ \texttt{"a"} +\!\!+ \Sigma^*$$

Automata-based approach

$$\frac{x \in R_1 \quad x \notin R_2}{x \in R_1 \cap comp(R_2)}$$

Problem

- Complement and interesection are **expensive**

Reasoning about Regular Expressions

Regular Expression Example

$$x \in [0..9]^* +\!\!+ \texttt{"a"} +\!\!+ \Sigma^* +\!\!+ \texttt{"b"} +\!\!+ \Sigma^*$$
$$x \notin [0..9]^* +\!\!+ \texttt{"a"} +\!\!+ \Sigma^*$$

Word-Based Approach

$$x = x_1 +\!\!+ \texttt{"a"} +\!\!+ x_2 +\!\!+ \texttt{"b"} +\!\!+ x_3 \land x_1 \in [0..9]^*$$
$$\forall\, x_4, x_5, x_6.\; x = x_4 +\!\!+ x_5 +\!\!+ x_6 \to x_4 \notin [0..9]^* \lor x_5 \neq \texttt{"a"}$$

Problem: Leads to a non-terminating sequence of unfoldings:

$$x_1 = \epsilon \lor x_1 \in [0..9] \lor (x_1 = x_7 +\!\!+ x_8 +\!\!+ x_9 \land x_7 \in [0..9] \land$$
$$x_8 \in [0..9]^* \land x_9 \in [0..9])$$

Reasoning about Regular Expressions

Regular Expression Example

$$x \in [0..9]^* \mathbin{+\!\!+} \texttt{"a"} \mathbin{+\!\!+} \Sigma^* \mathbin{+\!\!+} \texttt{"b"} \mathbin{+\!\!+} \Sigma^*$$
$$x \notin [0..9]^* \mathbin{+\!\!+} \texttt{"a"} \mathbin{+\!\!+} \Sigma^*$$

Word-based approach with incomplete procedures

$$\frac{x \in R_1 \quad x \notin R_2 \quad L(R_1) \subseteq L(R_2)}{\text{UNSAT}}$$

- Use fast, incomplete procedure to justify $L(R_1) \subseteq L(R_2)$

Proving $L(R_1) \subseteq L(R_2)$

$$\frac{}{L(\epsilon) \subseteq L(R)} \qquad \frac{}{L(R) \subseteq L(\Sigma^*)} \qquad \frac{\forall\, x \in L(R).\ |x| = 1}{L(R) \subseteq L(\Sigma)}$$

$$\frac{L(R_1) \subseteq L(R_2) \quad L(R_2) \subseteq L(R_3)}{L(R_1) \subseteq L(R_3)} \qquad \frac{}{L(R) \subseteq L(R^*)} \qquad \frac{L(R_1) \subseteq L(R_2)}{L(R_1^*) \subseteq L(R_2^*)}$$

$$\frac{L(R_1) \subseteq L(R_2)}{L(R \mathbin{+\!\!+} R_1) \subseteq L(R \mathbin{+\!\!+} R_2)} \qquad \frac{c_1 \geq c_3 \quad c_2 \leq c_4}{L([c_1..c_2]) \subseteq L([c_3..c_4])}$$

Reasoning about Regular Expressions

Regular Expression Example

$$x \in [0..9]^* \mathbin{+\!\!+} \texttt{"a"} \mathbin{+\!\!+} \Sigma^* \mathbin{+\!\!+} \texttt{"b"} \mathbin{+\!\!+} \Sigma^*$$
$$x \notin [0..9]^* \mathbin{+\!\!+} \texttt{"a"} \mathbin{+\!\!+} \Sigma^*$$

Reasoning about Language Inclusion

$$\dfrac{\dfrac{}{L(\Sigma^* \mathbin{+\!\!+} \texttt{"b"} \mathbin{+\!\!+} \Sigma^*) \subseteq L(\Sigma^*)}}{L([0..9]^* \mathbin{+\!\!+} \texttt{"a"} \mathbin{+\!\!+} \Sigma^* \mathbin{+\!\!+} \texttt{"b"} \mathbin{+\!\!+} \Sigma^*) \subseteq L([0..9]^* \mathbin{+\!\!+} \texttt{"a"} \mathbin{+\!\!+} \Sigma^*)}$$

More Information

Strings Papers

- "A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions" by Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. In Proceedings of the 26^{th} International Conference on Computer Aided Verification (CAV '14), (Armin Biere and Roderick Bloem, eds.), July 2014, pp. 646-662. Vienna, Austria.
- "An Efficient SMT Solver for String Constraints" by Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. Formal Methods in System Design, vol. 48, no. 3, June 2016, pp. 206-234, Springer US.
- "Scaling up DPLL(T) String Solvers Using Context-Dependent Simplification" by Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. In Proceedings of the 29^{th} International Conference on Computer Aided Verification (CAV '17), (Rupak Majumdar and Viktor Kuncak, eds.), July 2017, pp. 453-474. Heidelberg, Germany.
- "High-Level Abstractions for Simplifying Extended String Constraints in SMT" by Andrew Reynolds, Andres Nötzli, Clark Barrett, and Cesare Tinelli. In Proceedings of the 31^{st} International Conference on Computer Aided Verification (CAV '19), (Isil Dillig and Serdar Tasiran, eds.), July 2019, pp. 23-42. New York, New York.
- "Even Faster Conflicts and Lazier Reductions for String Solvers" by Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Clark Barrett, and Cesare Tinelli. In Proceedings of the 34^{th} International Conference on Computer Aided Verification (CAV '22), (Sharon Shoham and Yakir Vizel, eds.), Aug. 2022, pp. 205-226. Haifa, Israel.

Amazon's Zelkova Tool

- J. Backes et al., "Semantic-based Automated Reasoning for AWS Access Policies using SMT," 2018 Formal Methods in Computer Aided Design (FMCAD), Austin, TX, 2018.