A Decision Procedure for String to Code Point Conversion

Andrew Reynolds
Cesare Tinelli

THE UNIVERSITY OF IOWA

Andres Noetzli
Clark Barrett

Stanford University

IJCAR 2020

July 3, 2020

SMT Solvers for Strings

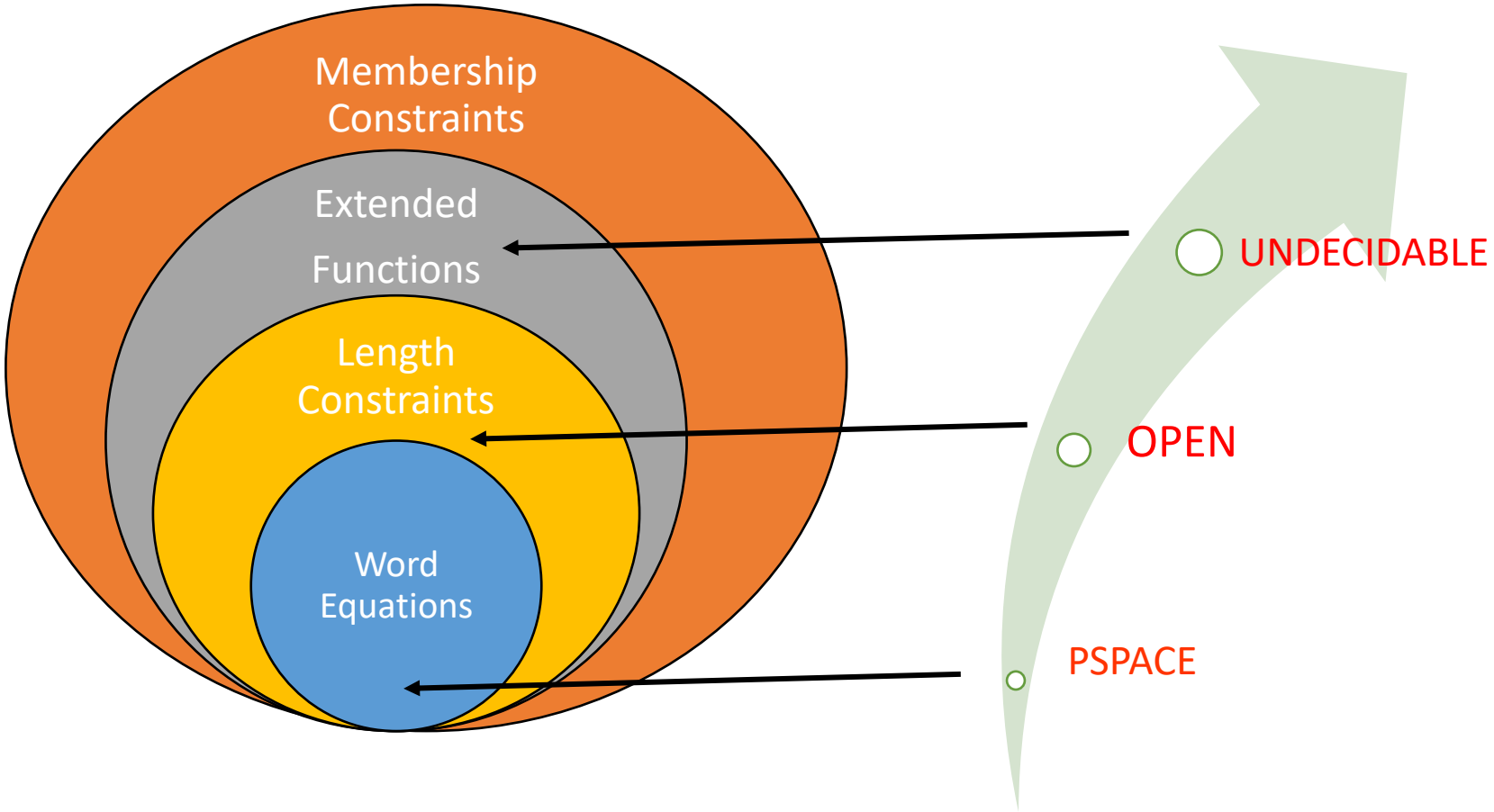
- Strings are important in formal methods applications:
  - SQL injections, XSS vulnerabilities in web applications
  - Reasoning about access policies in cloud applications
  - Data processing

- In the past decade, efficient SMT solvers for strings have emerged
  - **cvc4**, Norn, S3 , Trau, Z3, Z3str, …

Overview

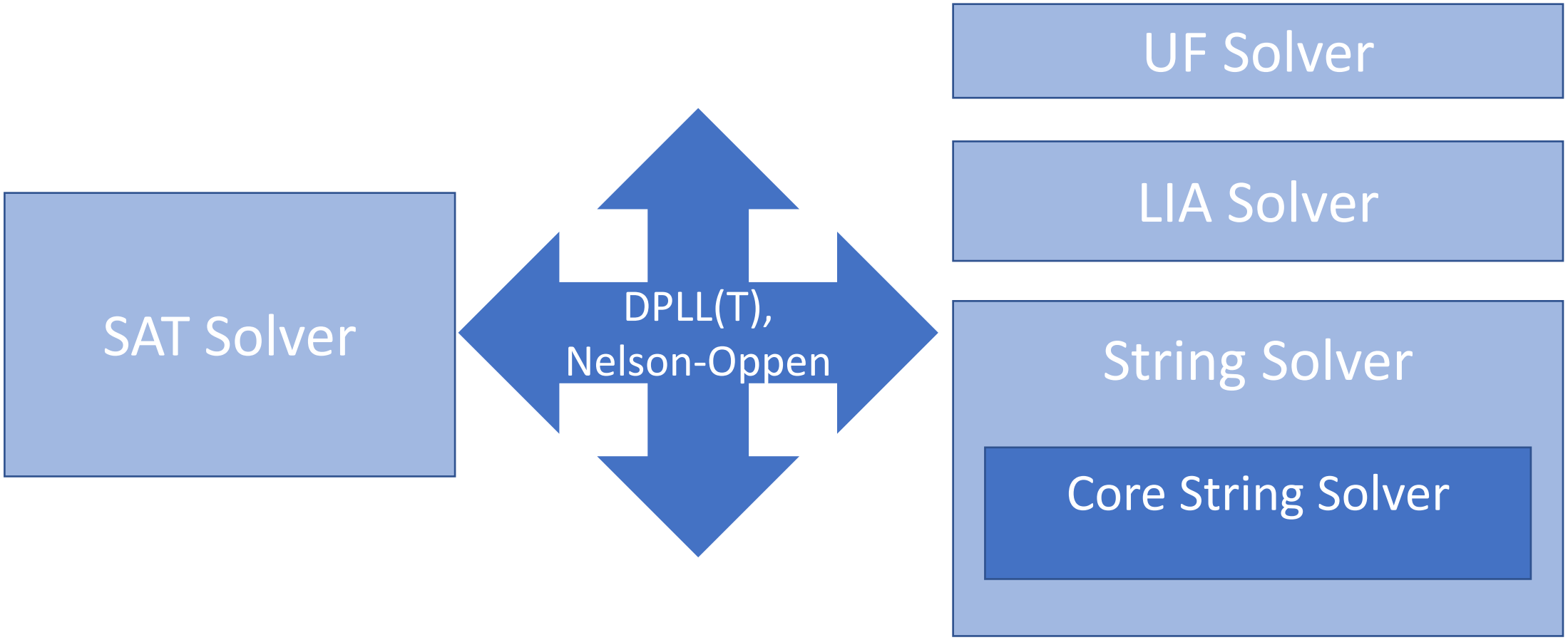
- A decision procedure for a simple theory of strings T_{AS}
  - Strings + length constraints + *string-to-code conversion*
- Improved reductions for extended string functions
- Implementation in CVC4
- Experimental Results
  - Significant improvement in state-of-the-art for SMT string solvers

Strings and RegExp: Theoretical Challenges



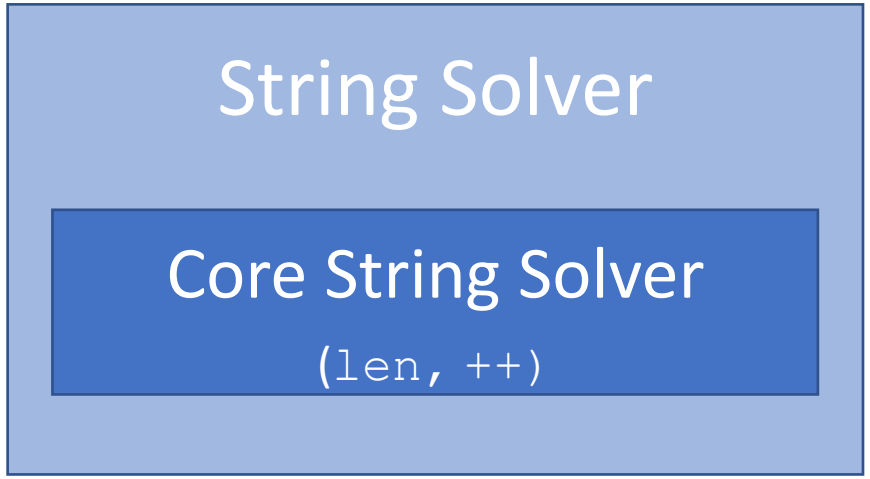
- Many applications require *extended string functions* and *RegEx memberships*
  - `to_int(x)≠44, to_lower(x)="abc", x∈range("A","Z")`

SMT Solvers for Strings + Extended Functions



SMT Solvers for Strings + Extended Functions

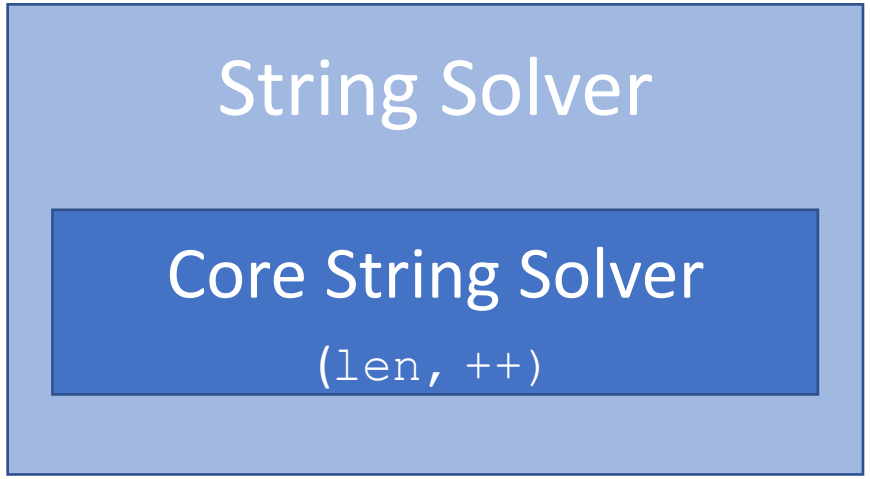
- *Core* string solver **[Liang et al CAV 2014]**
  - (Dis)equalities over strings:
    - Variables, constants, concatenation `++`
  - (Dis)equalities with string length `len`



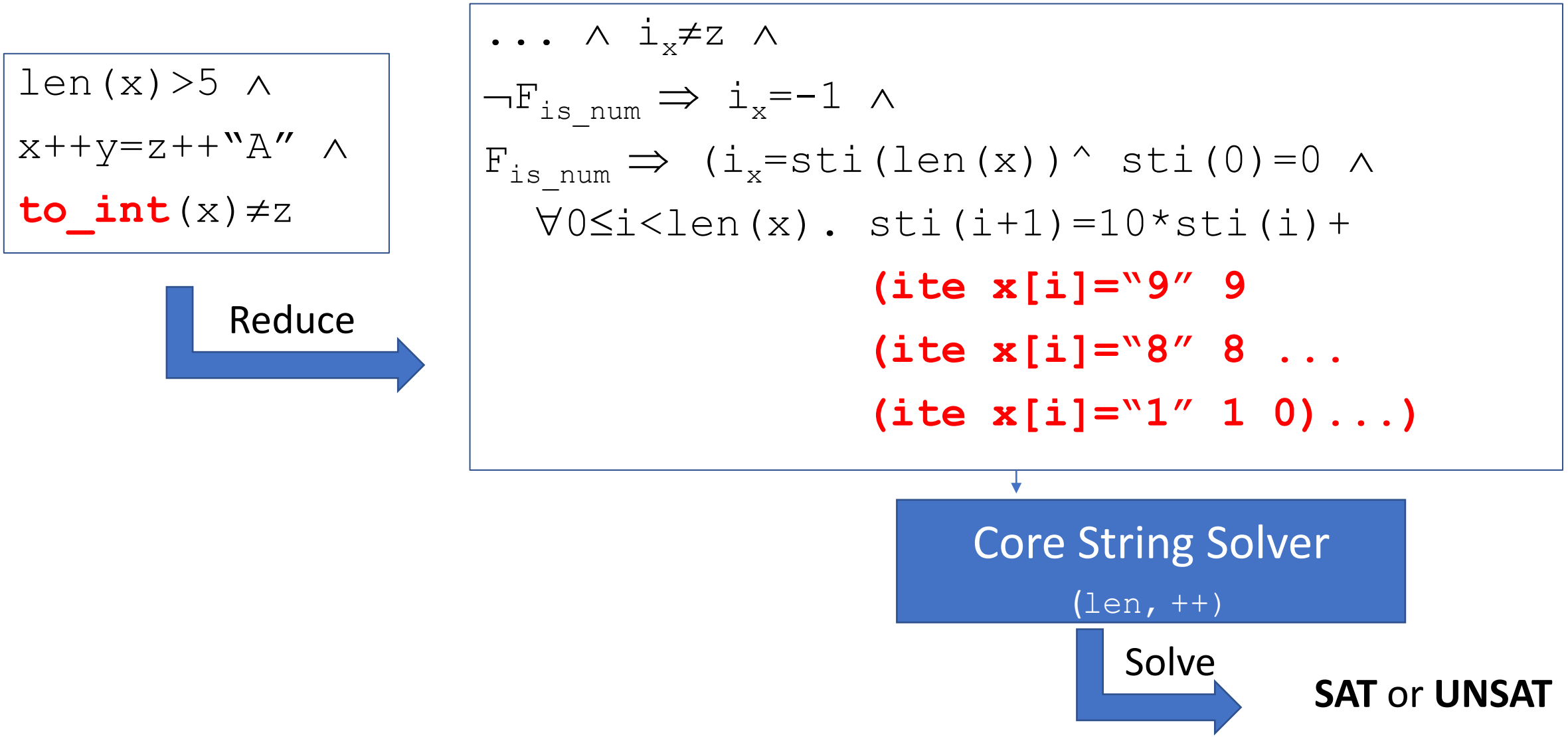
SMT Solvers for Strings + Extended Functions

```
len(x)>5 ∧
x++y=z++"A" ∧
to_int(x)≠z
```

- Extended functions can be reduced to word equations
  [Reynolds et al CAV 2017]



SMT Solvers for Strings + Extended Functions



SMT Solvers for Strings + Extended Functions (via code)

$\texttt{len(x)>5} \wedge$
$\texttt{x++y=z++“A”} \wedge$
$\texttt{to\_int(x)}\neq\texttt{z}$

Reduce

$\ldots \wedge i_x \neq z \wedge$
$\neg F_{is\_num} \Rightarrow i_x = -1 \wedge$
$F_{is\_num} \Rightarrow (i_x = sti(len(x)) \wedge sti(0)=0 \wedge$
$\forall 0 \leq i < len(x).\ sti(i+1) = 10*sti(i) +$
(code(x[i])-48)

- **In this paper:**
  - New procedure for string-to-code point
  - Enables efficient reductions for constraints in common applications

Core String Solver
(len, ++, ***code***)

Solve

SAT or **UNSAT**

Signature of Theory of Strings Σ_{AS}

Sorts

```
Int        Str        Bool
```

Σ_A

```
n  :  Int                          for all n∈Z
+  :  Int x Int  →  Int
-  :  Int  →  Int
≥  :  Int x Int  →  Bool
```

Σ_S

```
l     :  Str                       for all l∈𝒜*
len   :  Str  →  Int
code  :  Str  →  Int
```

… (Dis)equalities $=, \neq$ over all sorts

$\Rightarrow$ Intentionally omit string concatenation

String to code operator code

- Assume ordering on characters of alphabet $\mathcal{A}$:
  - $c_1 < \ldots < c_{|\mathcal{A}|-1}$
  - For each c_i, we call i its code point
- $\texttt{code} : \texttt{Str} \rightarrow \texttt{Int}$ is interpreted as:
  1. For w in $\mathcal{A}^1$, $\texttt{code}(w)$ is the code point of the single character in w
  2. For all other w, $\texttt{code}(w)$ is -1

- Notes:
  - In practice, we are interested in reasoning about Unicode strings
    - E.g. $\texttt{code}(\text{“A”}) = 65$, $\texttt{code}(\text{“5”}) = 53$
  - Alphabet can have any cardinality (≥ 1)

Rule-based calculus for strings

- Rules operate on configurations <A, S> where:
  - **A:** a set of arithmetic constraints
  - **S:** a set of string constraints
- Each rule in guarded assignment form:

$$\textbf{RuleName}\ \frac{\text{required conditions}}{\text{modifications to the configuration}} \quad \text{or} \quad \frac{\text{required conditions}}{\textbf{“unsat”}}$$

- Some rules have multiple conclusions, delimited by “||”:

$$\frac{\text{required conditions}}{\text{modification \#1} \quad || \quad \ldots \quad || \quad \text{modification \#n}}$$

Rule-based calculus for strings

- A run of the procedure generates a ***derivation tree***:
  - Nodes are configurations
  - Child nodes are obtained by legally applying a rule to its parent

- Given derivation tree with root <A, S>
  - A closed derivation tree (where all leaves are "**unsat**")

    $\Rightarrow$ <A, S> is T_{AS}-unsat
  - A derivation tree with a saturated configuration

    $\Rightarrow$ <A, S> is T_{AS}-sat

Core rules

$$\text{A-Conf}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \bot}{\text{unsat}} \qquad \text{A-Prop}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} s \approx t \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{S} := \mathsf{S}, s \approx t}$$

$$\text{S-Conf}\ \frac{\mathsf{S} \models \bot}{\text{unsat}} \qquad \text{S-Prop}\ \frac{\mathsf{S} \models s \approx t \quad s,t : \mathsf{Int} \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{A} := \mathsf{A}, s \approx t}$$

$$\text{L-Intro}\ \frac{\mathsf{S} \models x \approx l \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S},\ \mathsf{len}(x) \approx (\mathsf{len}(l))\downarrow} \qquad \text{L-Valid}\ \frac{x \in \mathcal{T}(\mathsf{A} \cup \mathsf{S}) \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S}, x \approx \epsilon \quad \| \quad \mathsf{A} := \mathsf{A},\ \mathsf{len}(x) > 0}$$

$$\text{Card}\ \frac{\mathsf{S} \models \mathsf{len}(x_1) \approx \ldots \approx \mathsf{len}(x_n) \quad n > 1}{\|_{1 \leqslant i < j \leqslant n}\ \mathsf{S} := \mathsf{S}, x_i \approx x_j \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(x_1) > \lfloor \log_{|\mathcal{A}|}(n-1) \rfloor}$$

- Rules adopted from previous work **[Liang et al CAV 2014]**

Core rules

$$\text{A-Conf}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \bot}{\text{unsat}} \qquad \text{A-Prop}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} s \approx t \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{S} := \mathsf{S}, s \approx t}$$

Models LIA solver :

- Conflicts
- Propagate Int equalities from A to S

$$\text{S-Conf}\ \frac{\mathsf{S} \models \bot}{\text{unsat}} \qquad \text{S-Prop}\ \frac{\mathsf{S} \models s \approx t \quad s,t : \mathsf{Int} \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{A} := \mathsf{A}, s \approx t}$$

$$\text{L-Intro}\ \frac{\mathsf{S} \models x \approx l \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S},\ \mathsf{len}(x) \approx (\mathsf{len}(l))\downarrow} \qquad \text{L-Valid}\ \frac{x \in \mathcal{T}(\mathsf{A} \cup \mathsf{S}) \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S}, x \approx \epsilon \quad \| \quad \mathsf{A} := \mathsf{A},\ \mathsf{len}(x) > 0}$$

$$\text{Card}\ \frac{\mathsf{S} \models \mathsf{len}(x_1) \approx \ldots \approx \mathsf{len}(x_n) \quad n > 1}{\|_{1 \leqslant i < j \leqslant n}\ \mathsf{S} := \mathsf{S}, x_i \approx x_j \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(x_1) > \lfloor \log_{|\mathcal{A}|}(n-1) \rfloor}$$

- Rules adopted from previous work **[Liang et al CAV 2014]**

Core rules

$$\text{A-Conf}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \bot}{\text{unsat}} \qquad \text{A-Prop}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} s \approx t \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{S} := \mathsf{S}, s \approx t}$$

$$\text{S-Conf}\ \frac{\mathsf{S} \models \bot}{\text{unsat}} \qquad \text{S-Prop}\ \frac{\mathsf{S} \models s \approx t \quad s,t : \mathsf{Int} \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{A} := \mathsf{A}, s \approx t}$$

Models core string solver :

- Conflicts by pure equality reasoning
- Propagate Int equalities from S to A

$$\text{L-Intro}\ \frac{\mathsf{S} \models x \approx l \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S},\ \mathsf{len}(x) \approx (\mathsf{len}(l))\downarrow} \qquad \text{L-Valid}\ \frac{x \in \mathcal{T}(\mathsf{A} \cup \mathsf{S}) \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S}, x \approx \epsilon \quad \| \quad \mathsf{A} := \mathsf{A},\ \mathsf{len}(x) > 0}$$

$$\text{Card}\ \frac{\mathsf{S} \models \mathsf{len}(x_1) \approx \ldots \approx \mathsf{len}(x_n) \quad n > 1}{\|_{1 \leqslant i < j \leqslant n}\ \mathsf{S} := \mathsf{S}, x_i \approx x_j \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(x_1) > \lfloor \log_{|\mathcal{A}|}(n-1) \rfloor}$$

- Rules adopted from previous work **[Liang et al CAV 2014]**

Core rules

$$\text{A-Conf}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \bot}{\text{unsat}} \qquad \text{A-Prop}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} s \approx t \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{S} := \mathsf{S}, s \approx t}$$

$$\text{S-Conf}\ \frac{\mathsf{S} \models \bot}{\text{unsat}} \qquad \text{S-Prop}\ \frac{\mathsf{S} \models s \approx t \quad s,t : \mathsf{Int} \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{A} := \mathsf{A}, s \approx t}$$

$$\text{L-Intro}\ \frac{\mathsf{S} \models x \approx l \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S},\ \mathsf{len}(x) \approx (\mathsf{len}(l))\downarrow} \qquad \text{L-Valid}\ \frac{x \in \mathcal{T}(\mathsf{A} \cup \mathsf{S}) \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S}, x \approx \epsilon \quad \| \quad \mathsf{A} := \mathsf{A},\ \mathsf{len}(x) > 0}$$

$$\text{Card}\ \frac{\mathsf{S} \models \mathsf{len}(x_1) \approx \ldots \approx \mathsf{len}(x_n) \quad n > 1}{\|_{1 \leqslant i < j \leqslant n}\ \mathsf{S} := \mathsf{S}, x_i \approx x_j \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(x_1) > \lfloor \log_{|\mathcal{A}|}(n-1) \rfloor}$$

Reasoning about length:

- Fixed for constant strings
- Length is non-negative

- Rules adopted from previous work **[Liang et al CAV 2014]**

Core rules

$$\text{A-Conf}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \bot}{\text{unsat}} \qquad \text{A-Prop}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} s \approx t \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{S} := \mathsf{S}, s \approx t}$$

$$\text{S-Conf}\ \frac{\mathsf{S} \models \bot}{\text{unsat}} \qquad \text{S-Prop}\ \frac{\mathsf{S} \models s \approx t \quad s,t : \mathsf{Int} \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{A} := \mathsf{A}, s \approx t}$$

$$\text{L-Intro}\ \frac{\mathsf{S} \models x \approx l \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S},\ \mathsf{len}(x) \approx (\mathsf{len}(l))\downarrow} \qquad \text{L-Valid}\ \frac{x \in \mathcal{T}(\mathsf{A} \cup \mathsf{S}) \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S}, x \approx \epsilon \quad \| \quad \mathsf{A} := \mathsf{A},\ \mathsf{len}(x) > 0}$$

$$\text{Card}\ \frac{\mathsf{S} \models \mathsf{len}(x_1) \approx \ldots \approx \mathsf{len}(x_n) \quad n > 1}{\|_{1 \leqslant i < j \leqslant n}\ \mathsf{S} := \mathsf{S}, x_i \approx x_j \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(x_1) > \lfloor \log_{|\mathcal{A}|}(n-1) \rfloor}$$

Reasoning about cardinality :

- Only finitely many strings exist for each length, since alphabet $\mathcal{A}$ is finite.

- Rules adopted from previous work **[Liang et al CAV 2014]**

Rules for code points (1/3)

$$\text{C-Intro}\ \frac{\mathsf{S} \models x \approx l \quad l \in \mathcal{A}^1}{\mathsf{A} := \mathsf{A}, \mathsf{code}(x) \approx (\mathsf{code}(l))\downarrow} \qquad \text{C-Collapse}\ \frac{\mathsf{S} \models x \approx l \quad \mathsf{code}(x) \in \mathcal{T}(\mathsf{S})}{\mathsf{A} := \mathsf{A}, \mathsf{code}(x) \approx (\mathsf{code}(l))\downarrow}$$

- Code of string `x` must be equal to its value, if it is forced by an equality
  - If x is equal to a constant string of length one:
    - `x="A"∈ S` … `code(x)=65 ∈ A`
  - If x is equal to any constant string, and code(x) is a term in S
    - `x="abc", code(x)=code(y)∈ S` … `code(x)=-1 ∈ A`

Rules for code points (2/3)

$$\text{C-Valid}\ \frac{\mathsf{code}(x) \in \mathcal{T}(\mathsf{S})}{\mathsf{A} := \mathsf{A}, \mathsf{len}(x) \not\approx 1, \mathsf{code}(x) \approx -1 \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(x) \approx 1, 0 \leqslant \mathsf{code}(x) < |\mathcal{A}|}$$

- For each string $\mathtt{x}$:
  - The length of $\mathtt{x}$ is not $\mathtt{1}$ and its code point is $-\mathtt{1}$
  - The length of $\mathtt{x}$ is $\mathtt{1}$ and its code point is in range $[\mathtt{0}\ldots|\mathcal{A}|-\mathtt{1}]$

Rules for code points (3/3)

$$\text{C-Inj}\ \frac{\mathsf{code}(x), \mathsf{code}(y) \in \mathcal{T}(\mathsf{S}) \quad x, y \text{ distinct}}{\mathsf{A} := \mathsf{A}, \mathsf{code}(x) \approx -1 \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{code}(x) \not\approx \mathsf{code}(y) \quad \| \quad \mathsf{S} := \mathsf{S}, x \approx y}$$

- Code is injective over the domain of strings of length 1
  - For each `code(x)`, `code(y)`, either:
  1. One of these strings, w.l.o.g. assume `x`, is not length `1` … `code(x)=-1`
  2. `x` and `y` have different code points … `code(x)≠code(y)`
  3. `x` and `y` are the same string … `x=y`

Example #1

$\langle A_0, S_0 \rangle$

- A = {`len(x)>len(y), code(x)=code(y), code(x)≥0`}
- S = {}

Example #1

$$\frac{\langle A_0, S_0 \rangle}{\mathbf{code(x)=code(y) \in S}}\ \text{A-Prop}$$

- A = {len(x)>len(y), code(x)=code(y), code(x)≥0}
- S = {code(x)=code(y)}

Example #1

$$\dfrac{\dfrac{\langle A_0, S_0\rangle}{\mathbf{code(x)=code(y)\in S}}\ \text{A-Prop}}{\texttt{code(x)=-1}\in\texttt{A} \qquad \texttt{code(x)}\neq\texttt{code(y)}\in\texttt{A} \qquad \texttt{x=y}\in\texttt{S}}\ \text{C-Inj}$$

- A = {$\texttt{len(x)>len(y)}$, $\texttt{code(x)=code(y)}$, $\texttt{code(x)}\geq 0$}
- S = {$\texttt{code(x)=code(y)}$}

Example #1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle A_0, S_0\rangle}}{\texttt{code(x)=code(y)}\in S}\ \text{A-Prop}}{\texttt{code(x)}\neq\texttt{code(y)}\in A}\quad \texttt{x=y}\in S}{\texttt{code(x)=-1}\in A}\ \text{C-Inj}}{\textbf{unsat}}\ \text{A-Conf}$$

- A = {len(x)>len(y), code(x)=code(y), code(x)≥0, code(x)=-1}
- S = {code(x)=code(y)}

Example #1

$$\dfrac{\dfrac{\dfrac{\langle A_0, S_0\rangle}{code(x)=code(y)\in S}\ \text{A-Prop}}{\dfrac{code(x)=-1\in A}{\textbf{unsat}}\ \text{A-Conf}} \quad \dfrac{\mathbf{code(x)\neq code(y)\in A}}{\textbf{unsat}}\ \text{A-Conf} \quad x=y\in S}{}\ \text{C-Inj}$$

- A = {$len(x)>len(y)$, $\underline{code(x)=code(y)}$, $code(x)\geq 0$, $\underline{code(x)\neq code(y)}$}
- S = {$code(x)=code(y)$}

Example #1

$$\dfrac{\dfrac{\dfrac{\langle A_0, S_0\rangle}{\texttt{code(x)=code(y)}\in S}\ \text{A-Prop}}{\texttt{code(x)=-1}\in A}\ \text{A-Conf}}{\textbf{unsat}} \qquad \dfrac{\dfrac{\texttt{code(x)}\neq\texttt{code(y)}\in A}{\textbf{unsat}}\ \text{A-Conf}}{\textbf{unsat}} \qquad \dfrac{}{\mathbf{x=y}\in S}\ \text{C-Inj}$$

- A = {len(x)>len(y), code(x)=code(y), code(x)≥0}
- S = {code(x)=code(y), x=y}

Example #1

$$\dfrac{\dfrac{\dfrac{\langle A_0, S_0\rangle}{\texttt{code(x)=code(y)}\in S}\ \text{A-Prop}}{\texttt{code(x)=-1}\in A}\ \text{A-Conf}\quad \dfrac{\texttt{code(x)}\neq\texttt{code(y)}\in A}{\textbf{unsat}}\ \text{A-Conf}}{\textbf{unsat}}\ \text{C-Inj}\qquad \dfrac{x=y\in S}{\textbf{len(x)=len(y)}\in \textbf{A}}\ \text{S-Prop}$$

- A = {len(x)>len(y), code(x)=code(y), code(x)≥0, len(x)=len(y)}
- S = {code(x)=code(y), x=y}

Example #1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle A_0, S_0\rangle}}{\texttt{code(x)=code(y)}\in S}\,\text{A-Prop}}{\texttt{code(x)=-1}\in A}\,\text{A-Conf}}{\textbf{unsat}} \quad \dfrac{\dfrac{\texttt{code(x)}\neq\texttt{code(y)}\in A}{}\,\text{A-Conf}}{\textbf{unsat}} \quad \dfrac{\dfrac{\dfrac{}{\texttt{x=y}\in S}\,\text{S-Prop}}{\texttt{len(x)=len(y)}\in A}\,\text{A-Conf}}{\textbf{unsat}}}{}\,\text{C-Inj}$$

- A = {$\underline{\texttt{len(x)>len(y)}}$, $\texttt{code(x)=code(y)}$, $\texttt{code(x)}\geq 0$, $\underline{\texttt{len(x)=len(y)}}$}
- S = {$\texttt{code(x)=code(y)}$, $\texttt{x=y}$}

Example #1

$\langle \mathtt{A}_0, \mathtt{S}_0 \rangle$

A-Prop

$\mathtt{code(x)=code(y)} \in \mathtt{S}$

C-Inj

$\mathtt{code(x)=-1} \in \mathtt{A}$

A-Conf

unsat

$\mathtt{code(x)} \neq \mathtt{code(y)} \in \mathtt{A}$

A-Conf

unsat

$\mathtt{x=y} \in \mathtt{S}$

S-Prop

$\mathtt{len(x)=len(y)} \in \mathtt{A}$

A-Conf

unsat

- Closed derivation tree … $\therefore$ $\mathtt{A}_0 \wedge \mathtt{S}_0$ is T_{AS}-unsatisfiable

Example #2

$\langle A_0, S_0 \rangle$

- A = $\{106 \geq code(x) \geq 97\}$
- S = $\{x \neq y, x \neq z, y = c_{97}, z = c_{106}\}$

Example #2

$$\frac{\langle A_0, S_0 \rangle}{\mathbf{code(y)=97, code(z)=106 \in A}}\ \text{C-Intro (2)}$$

- A = {106≥code(x)≥97, code(y)=97, code(z)=106}
- S = {x≠y, x≠z, y=c_{97}, z=c_{106}}

Example #2

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle A_0, S_0\rangle}}{\texttt{code(y)=97, code(z)=106}\in S}\ \text{C-Intro (2)}}{\dots}}{\textbf{unsat}} \qquad \dfrac{\dfrac{\texttt{code(x)}\neq\texttt{code(y)}\in A}{\dots}}{\textbf{unsat}}\ \text{C-Inj} \qquad \dfrac{\dfrac{\dots}{\textbf{unsat}}}{\dots}\ \text{C-Inj}}{\textbf{code(x)}\neq\textbf{code(z)}\in\textbf{A}} \qquad \dfrac{\dots}{\textbf{unsat}}$$

- A = $\{106 \geq \texttt{code(x)} \geq 97, \texttt{code(y)}=97, \texttt{code(z)}=106, \texttt{code(x)} \neq \texttt{code(y)}, \texttt{code(x)} \neq \texttt{code(z)}\}$
- S = $\{x \neq y,\ x \neq z,\ y = c_{97},\ z = c_{106}\}$

Example #2

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle A_0, S_0\rangle}}{\texttt{code(y)=97, code(z)=106}\in S}\ \text{C-Intro (2)}}{\dfrac{\dots}{\textbf{unsat}} \quad \texttt{code(x)}\neq\texttt{code(y)}\in A \quad \dfrac{\dots}{\textbf{unsat}}}\ \text{C-Inj}}{\dfrac{\dots}{\textbf{unsat}} \quad \textbf{code(x)}\neq\textbf{code(z)}\in\textbf{A} \quad \dfrac{\dots}{\textbf{unsat}}}\ \text{C-Inj}}{}\ \text{C-Inj}$$

- Derivation tree with saturated configuration … ∴ $A_0 \wedge S_0$ is T_{AS}-satisfiable
  - Model for `x` determined by model for `code(x)`

Correctness Theorem

- Calculus is:
  - **Terminating** (all derivation trees are finite)
  - **Refutation-sound**
    - A closed derivation tree with root M $\Rightarrow$ M is T_{AS}-unsat
  - **Solution-sound**
    - A derivation tree with root M with a saturated configuration $\Rightarrow$ M is T_{AS}-sat
      - Proof is constructive, a model for M can be produced in all such cases

$\therefore$ Calculus is a decision procedure for T_{AS}

Implementation Details

- Implemented in CVC4 SMT solver
  - As part of its DPLL(T) solver for strings

- Combined with other procedures for strings
  - E.g. **[Liang et al CAV 2014, Liang et al FroCoS 2015, Reynolds et al CAV 2017]**
  - All rules of calculus with premises $S \models l$ become $S \models_T l$
    - Where $\models_T$ denotes entailment modulo theory T

- Strategy crafted in these combinations:
  - C-Valid, C-Collapse applied eagerly
  - C-Intro, C-Inj applied lazily

Reductions: Conversion Functions

- More efficient reductions that leverage `code`, including:

- Conversion between strings and integers `to_int(x)`:
  - ⊗ `… ite(x[i]="9",9,ite(x[i]="8",8, …ite(x[i]="0",0,-1)…)`
  - ⇒ `… ite(48≤code(x[i])≤57,code(x[i])-48, -1)`

- Conversion between lowercase and uppercase strings `to_lower(x)`:
  - ⊗ `… ite(x[i]="A","a",ite(x[i]="B","b", … ite(x[i]="Z","z",x[i])…)`
  - ⇒ `… code(x[i])+ite(65≤code(x[i])≤90,32,0)`

Reductions: Ordering, RegExp Range

- Lexicographic ordering:
  - ⊗ $x \le y \Leftrightarrow \exists i \ldots (x[i]=y[i] \vee (x[i]=\text{“A”} \wedge y[i]=\text{“B”}) \vee (x[i]=\text{“A”} \wedge y[i]=\text{“C”}) \ldots)$
  - ⇒ $x \le y \Leftrightarrow \exists i \ldots \mathbf{code}(x[i]) \le \mathbf{code}(y[i])$

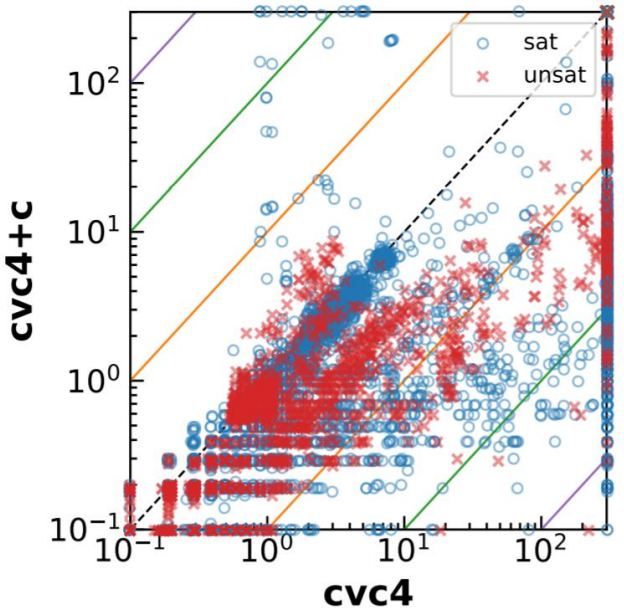
- Regular expression ranges:
  - ⊗ $x \in \text{range}(c_1, c_2) \Leftrightarrow \text{len}(x)=1 \wedge (x=c_1 \vee \ldots \vee x=c_2)$
  - ⇒ $x \in \text{range}(c_1, c_2) \Leftrightarrow \mathbf{code}(c_1) \le \mathbf{code}(x) \le \mathbf{code}(c_2)$

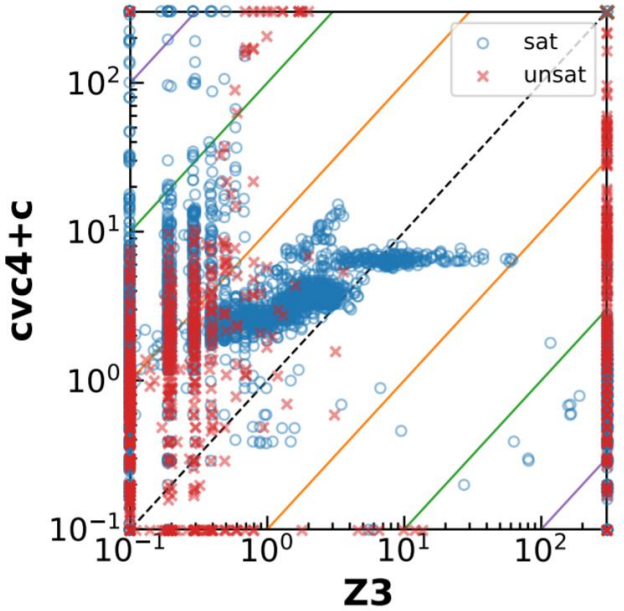
Experiments

- Compared solvers
  - **cvc4+c** : new version, includes techniques from this paper
  - **cvc4:** previous version
  - **z3**
- 21573 benchmarks from PyConByte
  - Correspond to concolic execution of Python code
  - Contain applications of `to_int`, `from_int`, regular expression ranges
- 300 second timeout

Experimental Results

Benchmark Set		cvc4+c	cvc4	z3
py-conbyte_cvc4	sat	**1344**	1104	1187
	unsat	**8576**	8547	8482
	×	13	282	264
py-conbyte_trauc	sat	**1009**	929	697
	unsat	1424	1407	**1428**
	×	13	110	321
py-conbyte_z3seq	sat	**1354**	1126	1343
	unsat	**5864**	5797	5719
	×	35	330	191
py-conbyte_z3str	sat	**711**	652	692
	unsat	**1227**	1223	1223
	×	3	66	26
Total	sat	**4418**	3811	3919
	unsat	**17091**	16974	16852
	×	64	788	802





- 10x t/o reduction
- Faster runtimes
- Improvement wrt state-of-the-art

Summary

- New decision procedure for string-to-code point conversion
- Efficient reduction techniques
  - Conversion functions, lexicographic ordering, regular expression ranges
- Implemented in CVC4
  - Combined with other procedures
- Experiments: significant improvement wrt SMT solvers for strings

Future Work

- Support for more extended functions of interest to user applications
- Proof production for strings