SMT, Strings, Security

Philipp Rümmer

Uppsala University

EPIT, April 10th, 2018

Plan

- String constraints by example
- A word equation primer
- Decidable fragments of string constraints

Strings in Verification

String in verification

```
// Pre = (true)
String s= '';
// P1 = (s ∈ ε)
while(*){
    // P2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
    s= 'a' + s + 'b';
}
// P3 = P2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P3
```

String in verification

ASCII, Unicode

```
// Pre = (true)
String s= '';
// P1 = (s ∈ ε)
while(*){
    // P2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
    s= 'a' + s + 'b';
}
// P3 = P2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P3
```

String in verification

Regular expression
assertion: $s = \epsilon$

// $Pre = (\quad)$

```
String s='';
```

// $P_1 = (s \in \epsilon)$

```
while(*){
```

// $P_2 = (s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v|)$

```
    s= 'a' + s + 'b';
}
```

// $P_3 = P_2$

```
assert(!s.contains('ba') && (s.length() % 2) == 0);
```

// $Post = P_3$

String in verification

```
// Pre = (true)
String s= '';
// P_1 = (s ∈ ε)
while(*){
      // P_2 = (s = u · v ∧ u ∈ a^* ∧ v ∈ b^* ∧ |u| = |v|)
      s= 'a' + s + 'b';
}
// P_3 = P_2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P_3
```

Word/string **concatenation**

String in verification

Loop invariant combining **word equations**, **regex** constraints, **length** constraints

```
// Pre = (true)
String s= '';
// P1 = (s ∈ ε)
while(*){
    // P2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
    s= 'a' + s + 'b';
}
// P3 = P2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P3
```

String in verification

```
// Pre = (true)
String s= '';
// P1 = (s ∈ ε)
while(*){
    // P2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
    s= 'a' + s + 'b';
}
// P3 = P2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P3
```

Substring constraint

String in verification

```
// Pre = (true)
String s= '';
// P_1 = (s ∈ ε)
while(*){
    // P_2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
    s= 'a' + s + 'b';
}
// P_3 = P_2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P_3
```

Or **regex:**

$s \notin \Sigma^* \cdot ba \cdot \Sigma^*$

String in verification

```
// Pre = (true)
String s= '';
// P1 = (s ∈ ε)
while(*){
    // P2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
    s= 'a' + s + 'b';
}
// P3 = P2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P3
```

Presburger length constraint

String in verification

```
// Pre = (true)
String s= '';
// P1 = (s ∈ ε)
while(*){
     // P2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
     s= 'a' + s + 'b';
}
// P3 = P2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P3
```

→ Need a solver that supports all those operators!

Alphabets

- All constraints are formulated w.r.t. to some fixed **finite alphabet** Σ
- $\Sigma = \{a, b, c, d\}$
- $\Sigma = \{0, \ldots, 255\}$ (e.g., 8-bit ASCII)
- $\Sigma = \{0, \ldots, 2^{32} - 1\}$ (e.g., UTF-32)

Semantics and notation

- Finite sequences of letters: Σ^*
- Empty word: ϵ
- Concatenation: $x \cdot y$

- Equations: $s = t$
- Regular expressions: $x \in \mathcal{L}$
- Word length: $|x|$

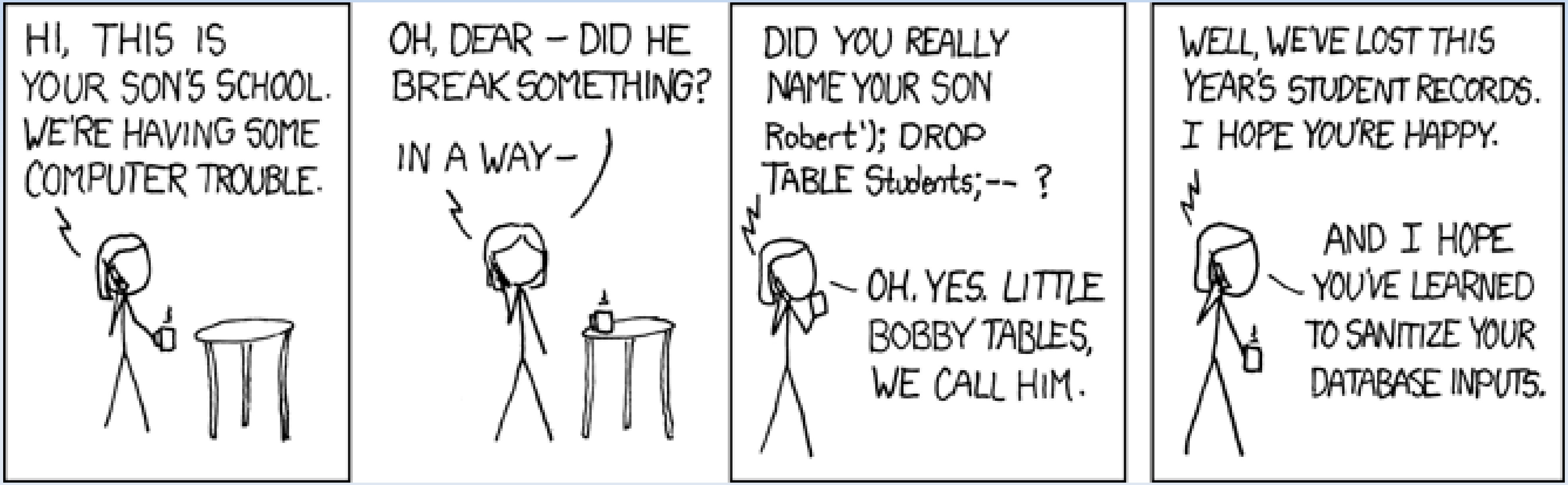
LARGE Alphabets

- Naive use of finite-state automata quickly becomes impossible
  - **Concrete** letters as transition guards → far too many transitions are needed to express interesting languages
  - **Symbolic** handling of letters is necessary

- Sometimes complex string conversion functions necessary, e.g.

  UTF-8 ↔ UTF-32

Injection attacks



xkcd.com

What is happening here?

Possible SQL command in a program

```
database.execute(
    "INSERT INTO students (name) VALUES ('"
    + name
    + "');");
```

What is happening here?

Possible SQL command in a program

```
database.execute(
    "INSERT INTO students (name) VALUES ('"
    + name
    + "');");
```

Command with input substituted

```
INSERT INTO students (name) VALUES ('Robert'); DROP TABLE students;--');
```

What is happening here?

Possible SQL command in a program

```
database.execute(
    "INSERT INTO students (name) VALUES ('"
    + name
    + "');");
```

Command with input substituted

```
INSERT INTO students (name) VALUES ('Robert'); DROP TABLE students;--');
```

Problem:
Input string
ends quotation!

Command
embedded in
user input
is executed

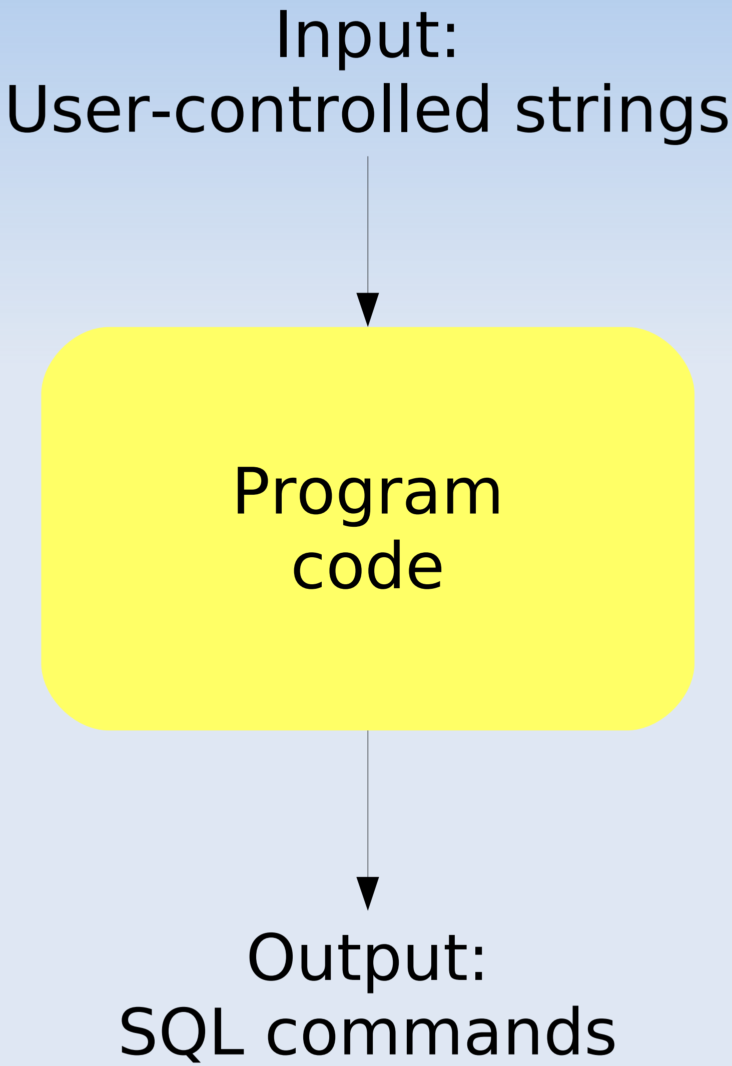
What is happening here?

Possible SQL command in a program

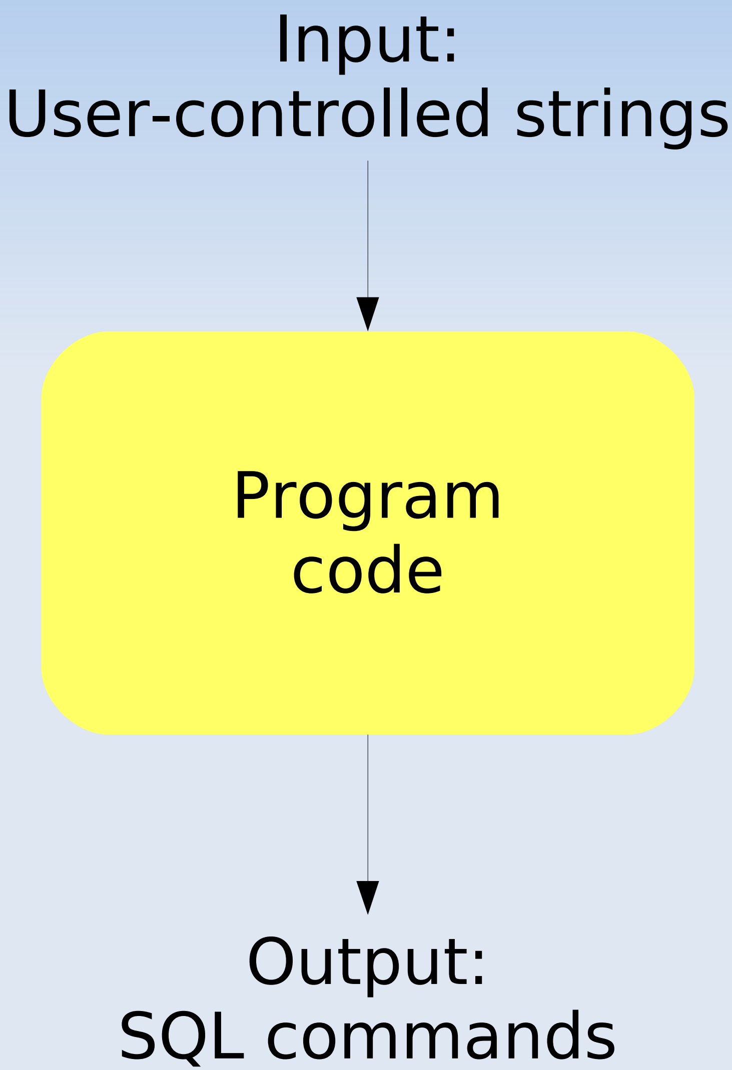
```
database.execute(
    "INSERT INTO students (name) VALUES ('"
    + name
    + "');");
```

- Since no **sanitisation** is applied, program is vulnerable to SQL injection attacks!

How can this be detected?



How can this be detected?



$$name \in \mathcal{L}_{input}$$

$$\wedge$$

$$\phi_{path}$$

$$\wedge$$

$$cmd \in \mathcal{L}_{attack}$$

How can this be detected?

Input:
User-controlled strings

↓

Program
code

↓

Output:
SQL commands

$$name \in \mathcal{L}_{input}$$

$$\wedge$$

$$\phi_{path}$$

$$\wedge$$

$$cmd \in \mathcal{L}_{attack}$$

Regex
or CFG

What is happening here?

Possible SQL command in a program

```
database.execute(
    "INSERT INTO students (name) VALUES ('"
    + name
    + "');");
```

- However, this case could more easily be found with techniques like **taint tracking**
- But what if sanitisation were actually applied?

A subtle XSS vulnerability

JavaScript embedded in a web-page

```
var x = goog.string.htmlEscape(cat);
var y = goog.string.escapeString(x);

catElem.innerHTML =
  '<button onclick="createCatList(\'' +
  y + '\')">' + x + '</button>';
```

A subtle XSS vulnerability

Input string

JavaScript embedded in a web-page

```
var x = goog.string.htmlEscape(cat);
var y = goog.string.escapeString(x);

catElem.innerHTML =
  '<button onclick="createCatList(\'' +
  y + '\')">' + x + '</button>';
```

A subtle XSS vulnerability

HTML escape: & → &

Input string

JavaScript escape: ' → \'

JavaScript embedded in a web-page

```
var x = goog.string.htmlEscape(cat);
var y = goog.string.escapeString(x);

catElem.innerHTML =
  '<button onclick="createCatList(\'' +
  y + '\')">' + x + '</button>';
```

A subtle XSS vulnerability

HTML escape:
& → &

Input string

JavaScript escape:
' → \'

JavaScript embedded in a web-page

```
var x = goog.string.htmlEscape(cat);
var y = goog.string.escapeString(x);

catElem.innerHTML =
  '<button onclick="createCatList(\'' +
  y + '\')">' + x + '</button>';
```

Implicit HTML **un**escape
of the `onclick`
attribute:
& → &

An XSS vulnerability (2)

JavaScript embedded in a web-page

```
var x = goog.string.htmlEscape(cat);
var y = goog.string.escapeString(x);

catElem.innerHTML =
  '<button onclick="createCatList(\'' +
  y + '\')">' + x + '</button>';
```

One possible attack

Choose `cat` to be `');alert(1);//`

Generated HTML string is then:

```
<button onclick="createCatList('&#39;);alert(1);//')">
&#39;);alert(1);//</button>
```

An XSS vulnerability (2)

JavaScript embedded in a web-page

```
var x = goog.string.htmlEscape(cat);
var y = goog.string.escapeString(x);

catElem.innerHTML =
    '<button onclick="createCatList(\'' +
    y + '\')">' + x + '</button>';
```

One possible atta

Choose `cat` to be `');`

This will be **unescaped** to
`createCatList('');alert(1);//')`

Generated HTML string is then:

```
<button onclick="createCatList('&#39;);alert(1);//')">
&#39;);alert(1);//</button>
```

An XSS vulnerability (2)

JavaScript embedded in a web-page

```
var x = goog.string.htmlEscape(cat);
var y = goog.string.escapeString(x);

catElem.innerHTML =
  '<button onclick="creat
  y + '\')">' + x + '</bu
```

Vulnerability since escape functions are applied in **wrong order**

One possible attack

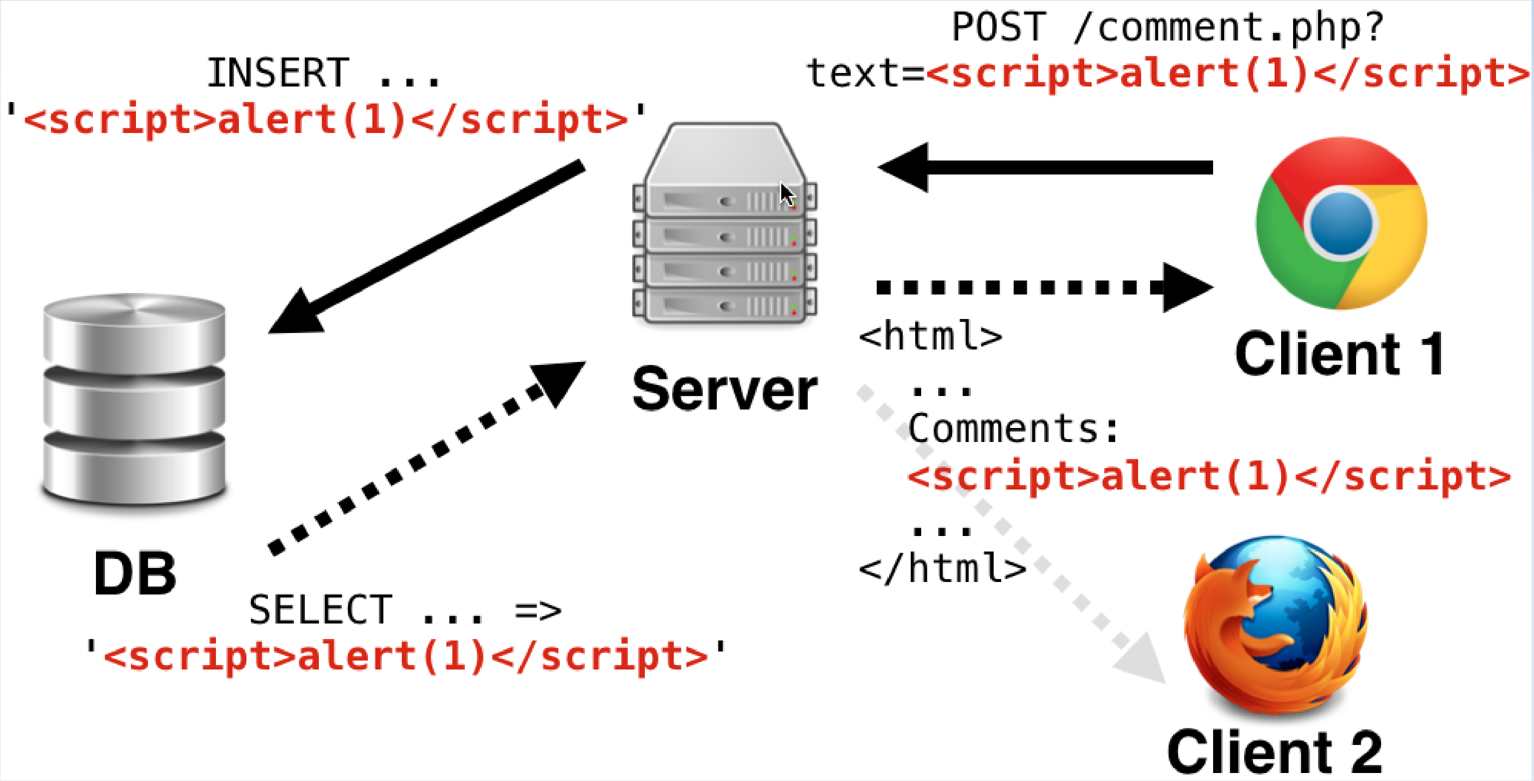
Choose `cat` to be `');`

This will be **unescaped** to `createCatList('');alert(1);//')`

Generated HTML string is then:

```
<button onclick="createCatList('&#39;);alert(1);//')">
&#39;);alert(1);//</button>
```

Cross-site scripting



http://blog.aboutme.vn/choi-xss-tai-knock-xss-moe/

Solvers for escape ops?

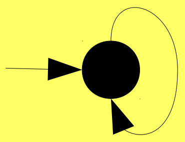
Solvers for escape ops?

- We need transducers!
  → Automata with multiple tracks

Solvers for escape ops?

- We need transducers!
  → Automata with multiple tracks

toUpperCase

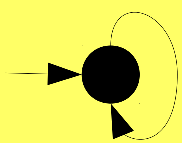


a/A
b/B
c/C
...

Solvers for escape ops?

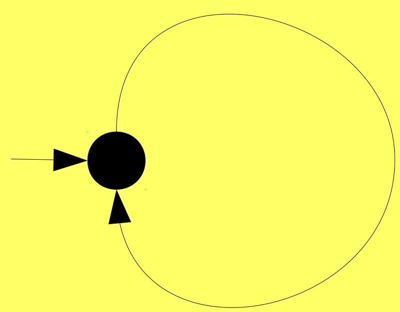
- We need transducers!
  → Automata with multiple tracks

toUpperCase



```
a/A
b/B
c/C
...
```

htmlEscape



```
</&lt;
>/&gt;
&/&amp;
...
```

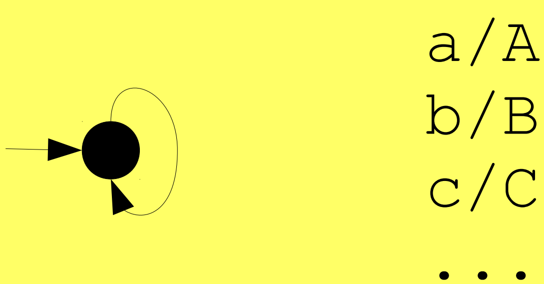
replaceAll

...

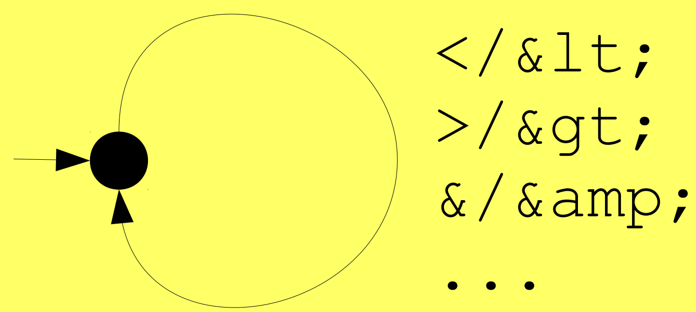
Solvers for escape ops?

- We need transducers!
  → Automata with multiple tracks

toUpperCase



htmlEscape



replaceAll

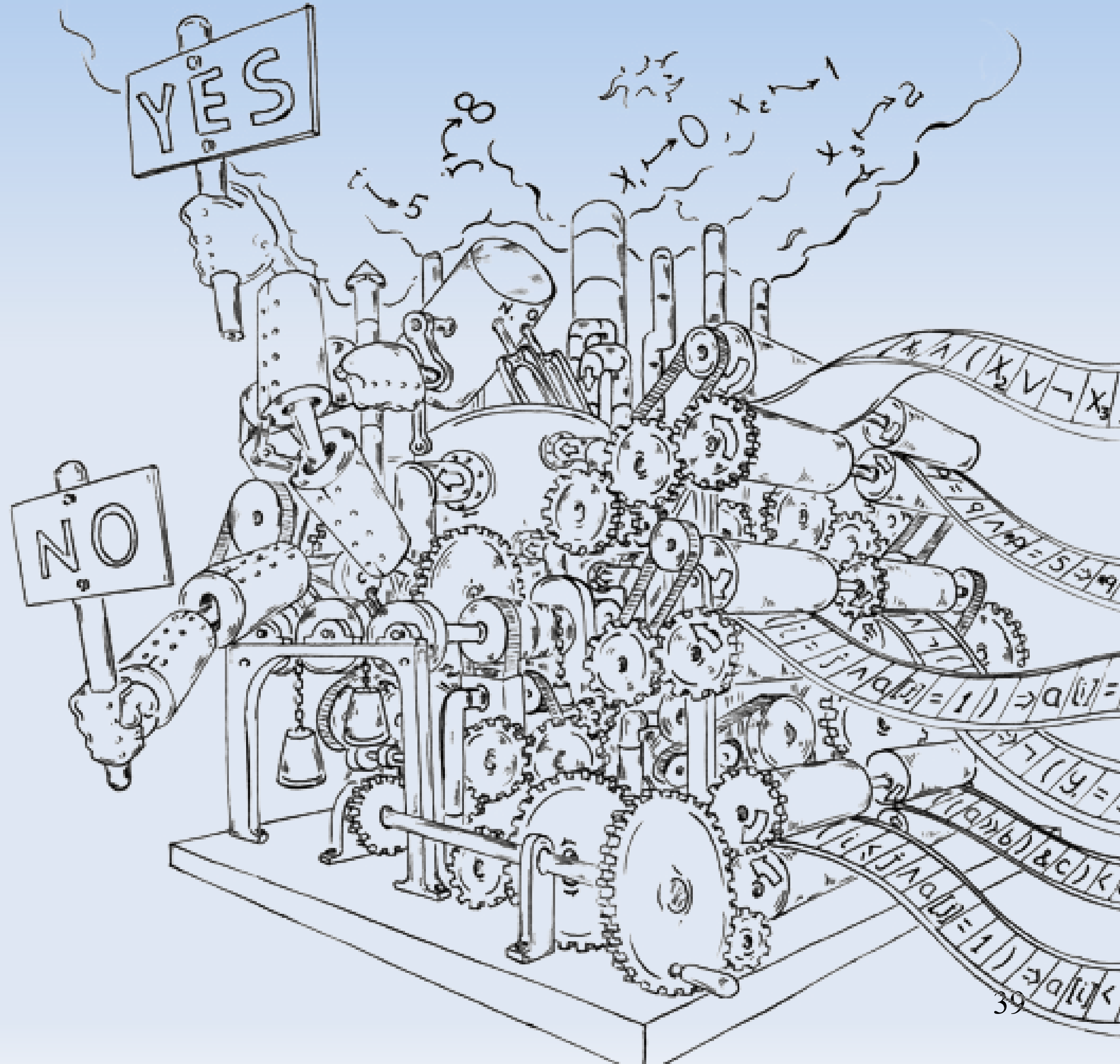
...

Do not preserve length ...

Other operations

- String reversal
- Context-free grammars
- String-to-number conversions
- Replace-all with symbolic arguments
- ...

Solving String Constraints



Bit of Solver History

- Bounded-length solvers
  - Bit-vector-based: Hampi, Kaluza
  - CP-based: Gecode
- Automata-based tools
  - Stranger, TRAU
- SMT/DPLL/CDCL-based methods
  - Z3-str/2/3, CVC4, S3/p, Norn, Sloth

(+ much theoretic work)

Solving Word Equations

- What are the solutions those equations?

$$s' = \text{`}a\text{'} \cdot s \cdot \text{`}b\text{'}$$

$$x \cdot y = u \cdot \text{`}ab\text{'} \cdot v$$

Nielsen's transformation

(also called Levi's lemma)

Theorem

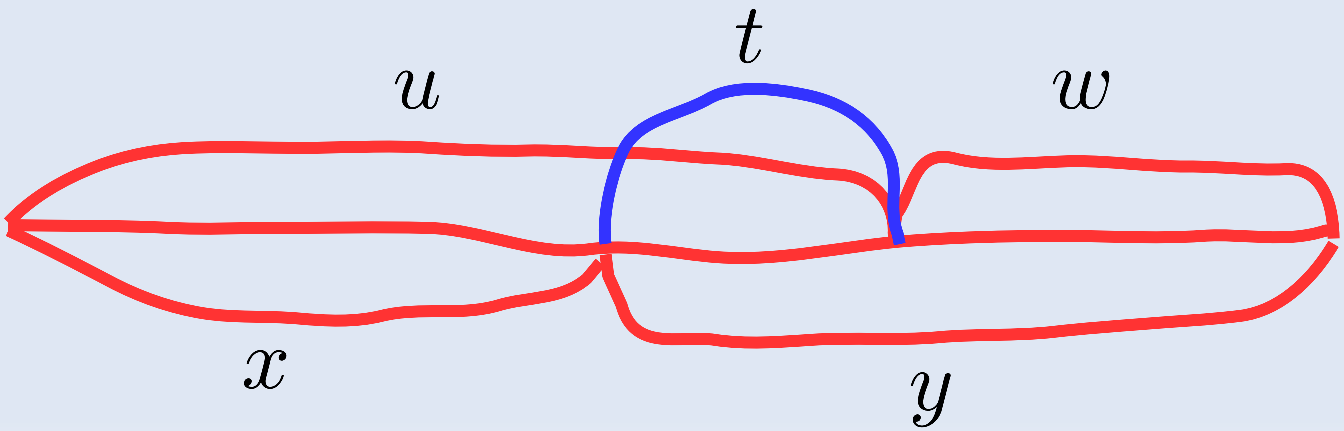
$$uw = xy \quad \Leftrightarrow \quad \begin{cases} \exists t. \;\; u = xt \wedge y = tw; \text{ or} \\ \exists t. \;\; x = ut \wedge w = ty \end{cases}$$

Nielsen's transformation

(also called Levi's lemma)

Theorem

$$uw = xy \quad \Leftrightarrow \quad \begin{cases} \exists t.\ \ u = xt \wedge y = tw;\ \text{or} \\ \exists t.\ \ x = ut \wedge w = ty \end{cases}$$



As a tableau rule

Nielsen's transformation

$$\frac{x\alpha = y\beta}{\begin{array}{c|c} x = yz & y = xz \\ z\alpha[x/yz] = \beta[x/yz] & \alpha[y/xz] = z\beta[y/xz] \end{array}}$$

$(z \text{ fresh})$

As a tableau rule

Nielsen's transformation

$$\frac{x\alpha = y\beta}{\begin{array}{c|c} x = yz & y = xz \\ z\alpha[x/yz] = \beta[x/yz] & \alpha[y/xz] = z\beta[y/xz] \end{array}}$$

$(z \text{ fresh})$

$$\frac{\alpha\beta = \alpha\gamma}{\beta = \gamma} \qquad \frac{\text{`}a\text{'}\alpha = \text{`}b\text{'}\beta}{*} \qquad \ldots$$

In the example

$$x \cdot y = u \cdot \text{`}ab\text{'} \cdot v$$

How about this one?

$$x \cdot y = y \cdot z$$

How about this one?

$$x \cdot y = y \cdot z$$

$$x = yt$$
$$ty = z$$

$$y = xt$$
$$xt = tz$$

How about this one?

$$x \cdot y = y \cdot z$$

$$x = yt$$
$$ty = z$$

$$y = xt$$
$$xt = tz$$

Cycle!

What can be done?

Ignore cycles and hope for the best!

Identify fragments for which NT is guaranteed to terminate

- Acyclic; straight-line

Improve NT and add termination criteria

- Makanin's method
- Simpler algorithms for **quadratic** equations

Quadratic word equations

Definition
A word equation is **quadratic** if each variable occurs at most twice in the equation.

- E.g.

$$x \cdot y = u \cdot \text{`}ab\text{'} \cdot v$$

$$x \cdot y = y \cdot z$$

- Consider satisfiability of a **single quadratic** equation

Quadratic = simpler?

Nielsen's transformation

$$\frac{x\alpha = y\beta}{\begin{array}{c|c} x = yz & y = xz \\ z\alpha[x/yz] = \beta[x/yz] & \alpha[y/xz] = z\beta[y/xz] \end{array}}$$

$(z \text{ fresh})$

Quadratic = simpler?

Nielsen's transformation

$$\frac{x\alpha = y\beta}{\begin{array}{c} x = yz \\ z\alpha[x/yz] = \beta[x/yz] \end{array} \quad \Big| \quad \begin{array}{c} y = xz \\ \alpha[y/xz] = z\beta[y/xz] \end{array}}$$

$(z \text{ fresh})$

Number of variable occurrences cannot increase!

A decision procedure

Modified Nielsen rule

$$\frac{x\alpha = y\beta}{\begin{array}{c|c|c} x \to y & x \to yz & y \to xz \\ \alpha[x/y] = \beta[x/y] & z\alpha[x/yz] = \beta[x/yz] & \alpha[y/xz] = z\beta[y/xz] \end{array}}$$

$(z \text{ fresh})$

A decision procedure

Modified Nielsen rule

$$\frac{x\alpha = y\beta}{\begin{array}{c|c|c} x \to y & x \to yz & y \to xz \\ \alpha[x/y] = \beta[x/y] & z\alpha[x/yz] = \beta[x/yz] & \alpha[y/xz] = z\beta[y/xz] \end{array}}$$

(z fresh)

Further rules

$$\frac{\alpha\beta = \alpha\gamma}{\beta = \gamma} \qquad \frac{\text{‘}a\text{’}\alpha = \text{‘}b\text{’}\beta}{*} \qquad \frac{\begin{array}{c}\alpha = \beta \\ \vdots \\ \alpha' = \beta'\end{array}}{*}$$

(equations equal up to renaming of variables)

Example

$$x \cdot \text{`}a\text{'} \cdot y = y \cdot \text{`}b\text{'} \cdot z$$

Even more rules

One-sided Nielsen rule

$$\frac{x\alpha = \text{`}a\text{'}\beta}{\begin{array}{c|c} x \to \epsilon & x \to \text{`}a\text{'}z \\ \alpha[x/\epsilon] = \text{`}a\text{'}\beta[x/\epsilon] & z\alpha[x/\text{`}a\text{'}z] = \beta[x/\text{`}a\text{'}z] \end{array}}$$

$(z \text{ fresh})$

Decision procedure?

Soundness

- If root is satisfiable, at least one branch cannot be closed

Completeness

- If root is unsat, a closed proof exists
- Follows from termination
- Open branches → satisfying assignments

Termination

- # of variable occurrences does not increase
- Up to renaming of variables, only finitely many different equations exist

Soundness argument

- Label equations $\alpha = \beta$ in the proof with:
  - $\top$ if equation is unsat
  - $\langle o, l \rangle$ if equation is sat, has o variable occurrences, and l is length of α for the shortest solution
- Order pairs $\langle o, l \rangle$ lexicographically

Lemma

In each application of the Nielsen rule, if the parent is labelled with $p < \top$, then at least one child has label $< p$.

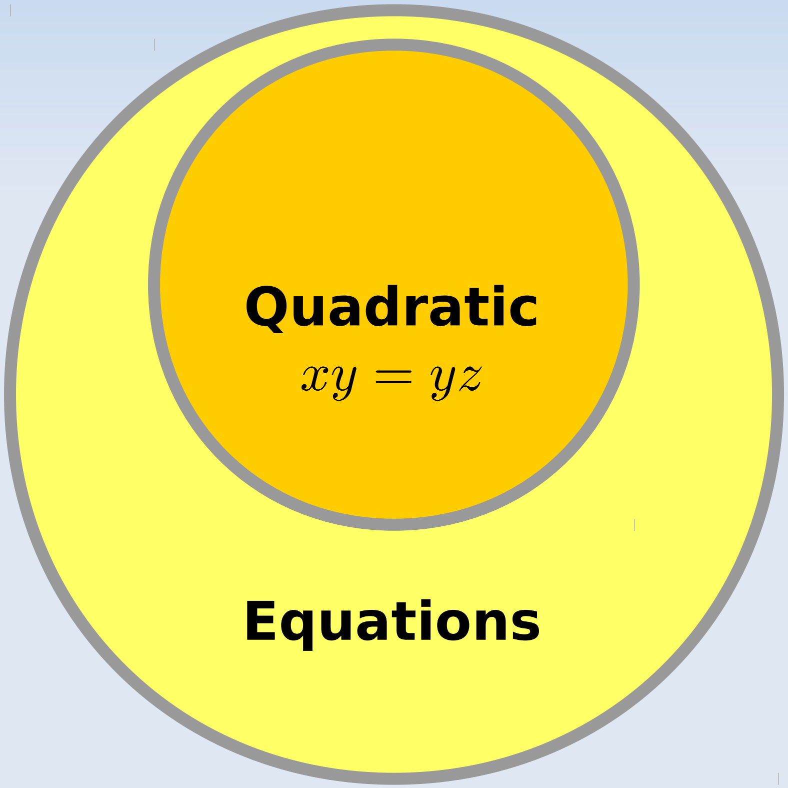
Soundness argument

- Label equations $\alpha = \beta$ in the proof with:
  - $\top$ if equation is unsat
  - $\langle o, l \rangle$ if equation is sat, has o variable occurrences, and l … r the shortest solution
- Order pairs $\langle o, l \rangle$ le…

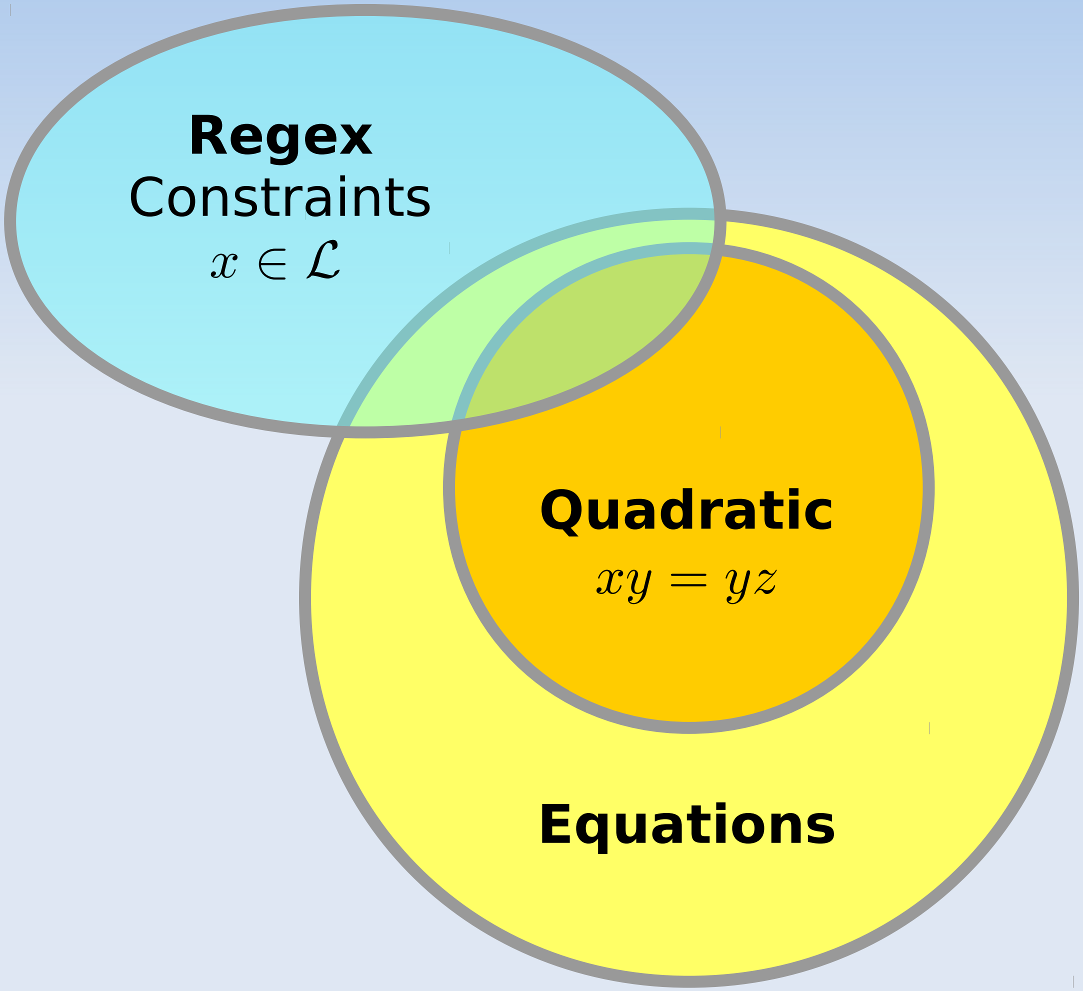
Decreasing labels → Branch cannot be closed!

Lemma
In each application of the Nielsen rule, if the parent is labelled with $p < \top$, then at least one child has label $< p$.

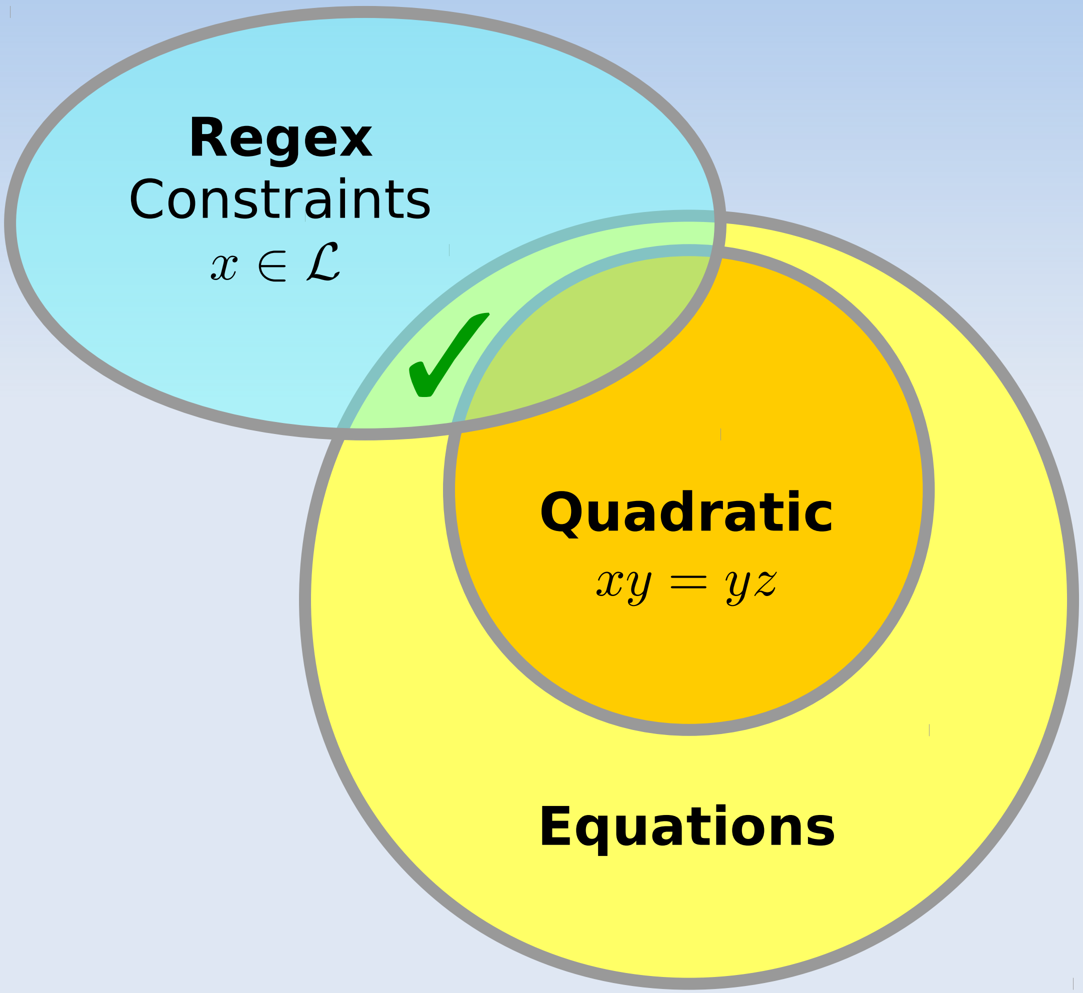
Combinations ...



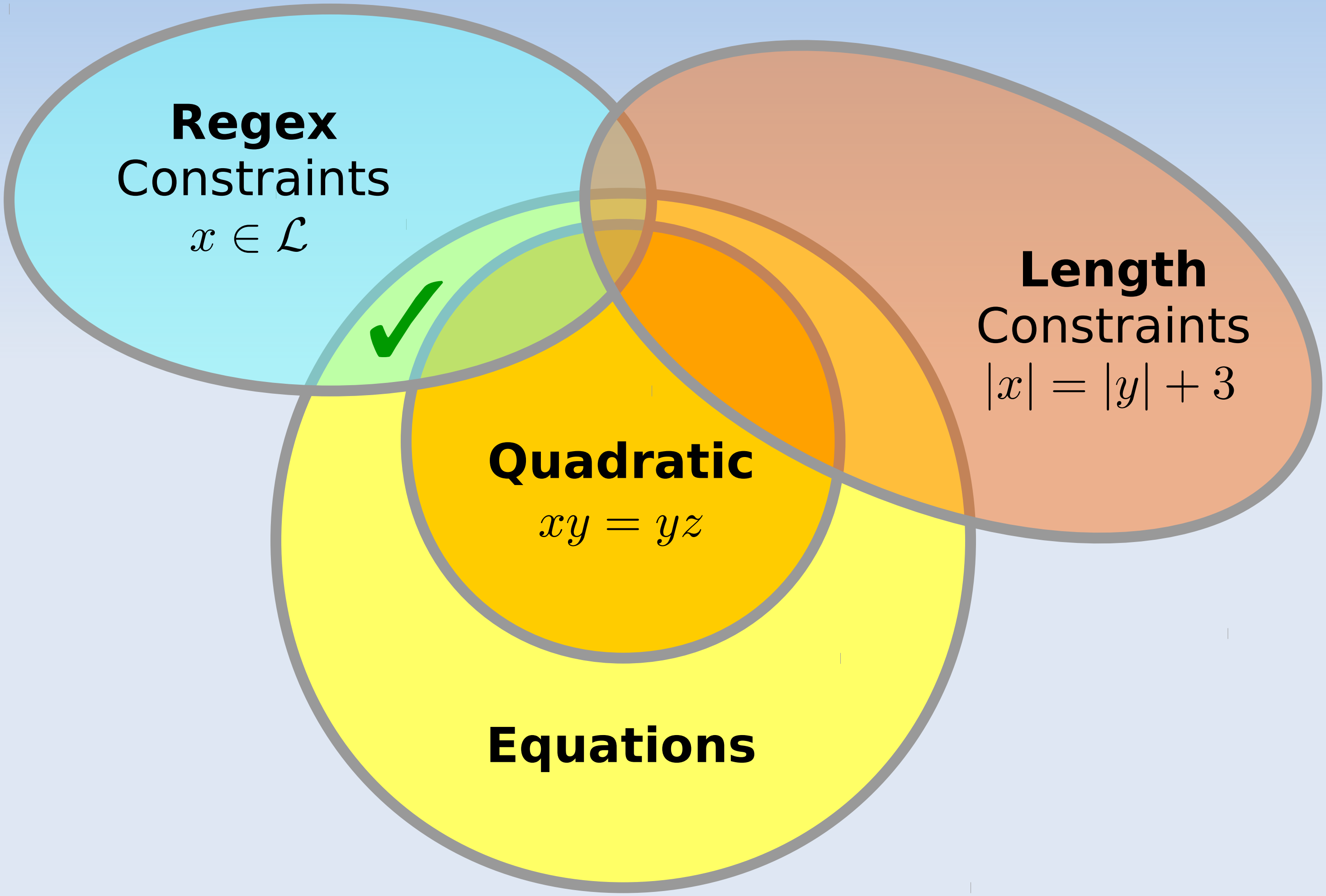
Combinations ...



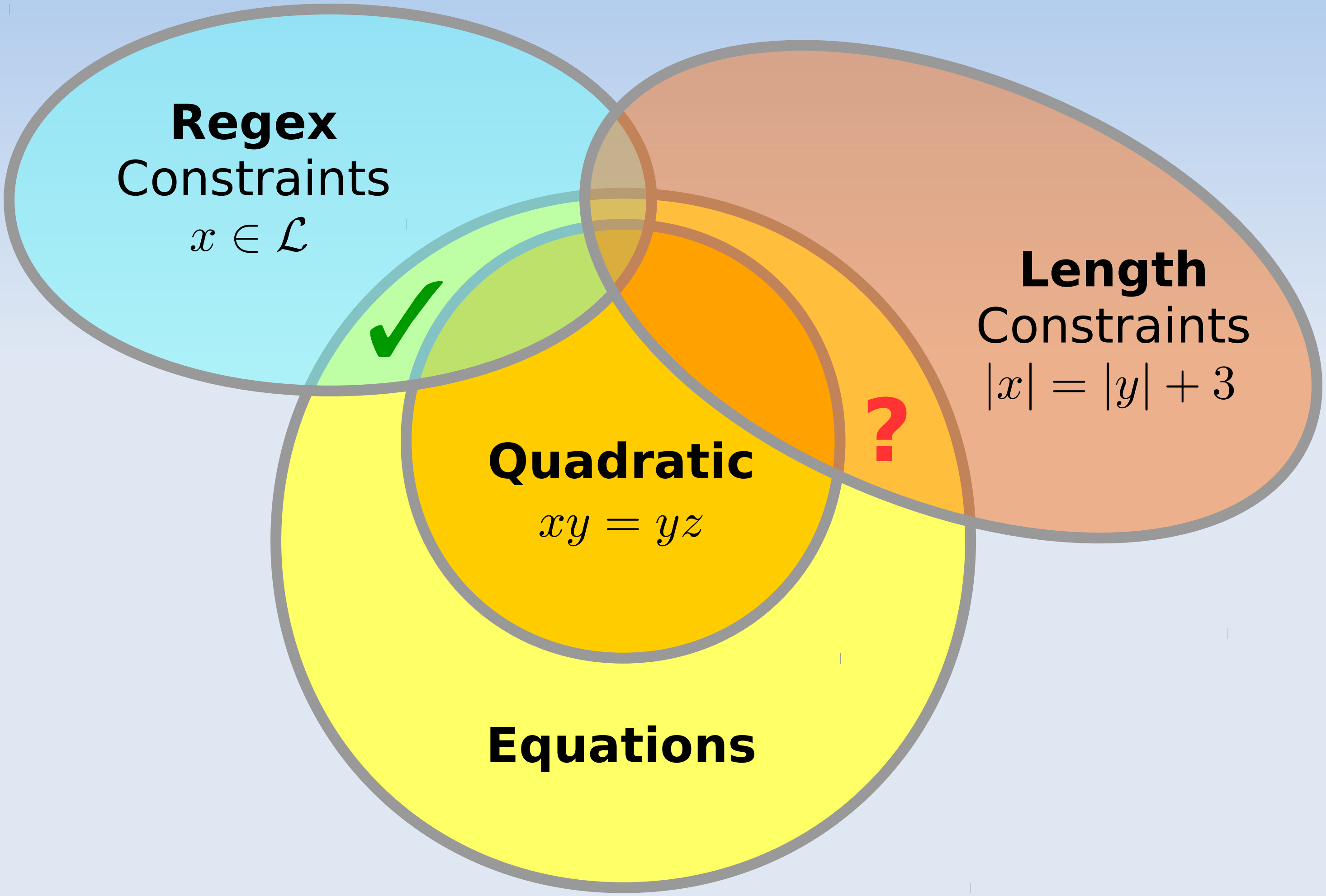
Combinations …



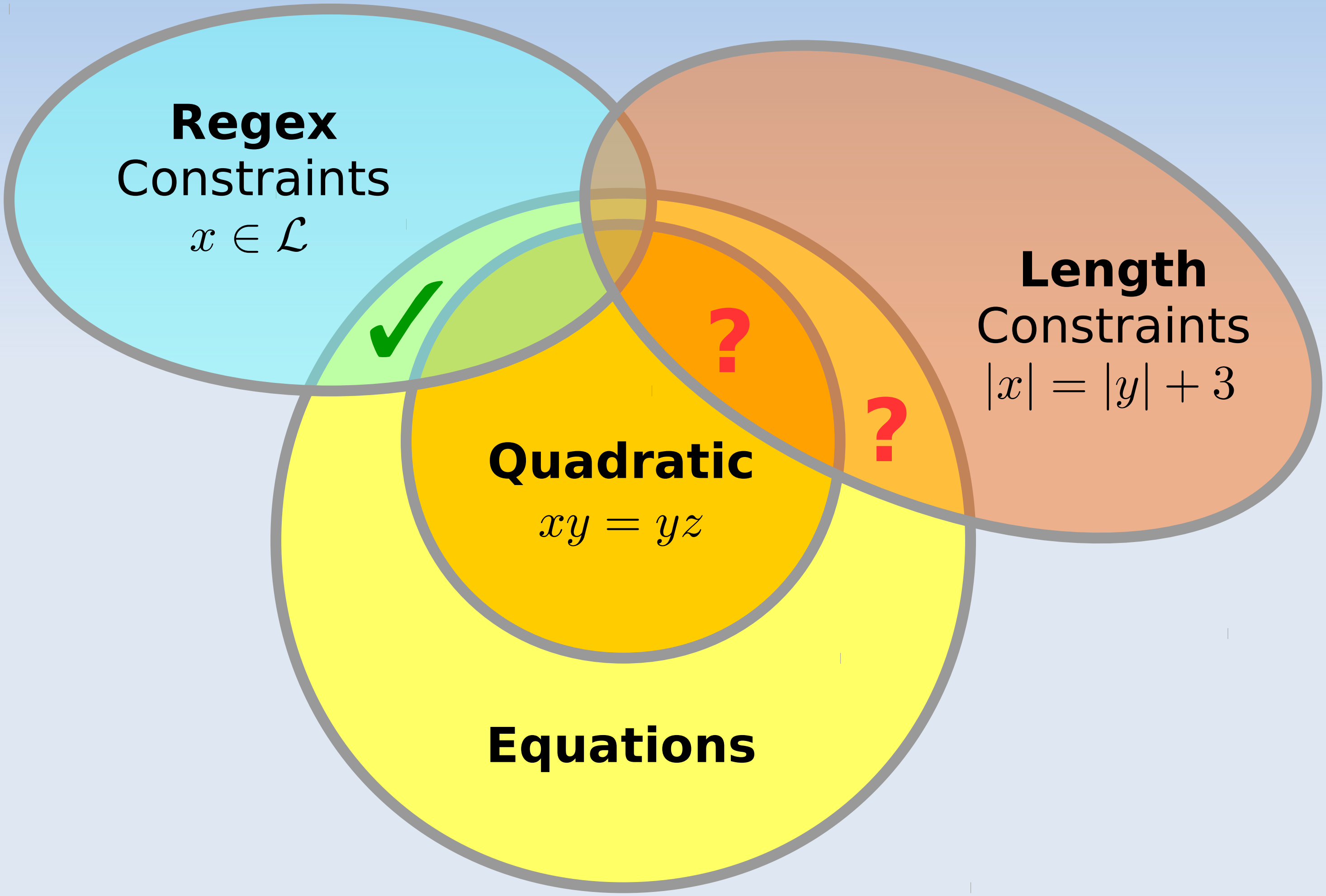
Combinations ...



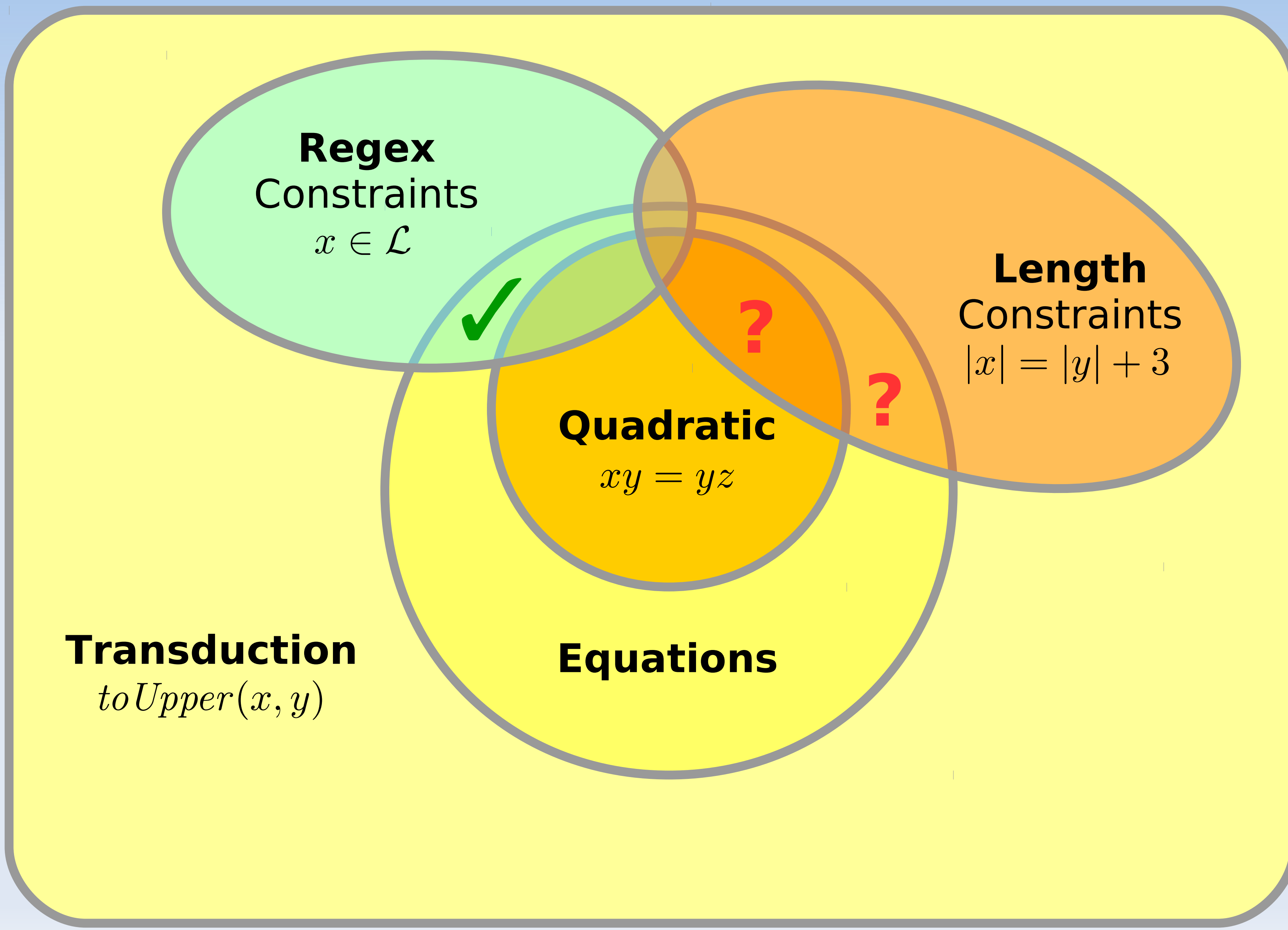
Combinations …



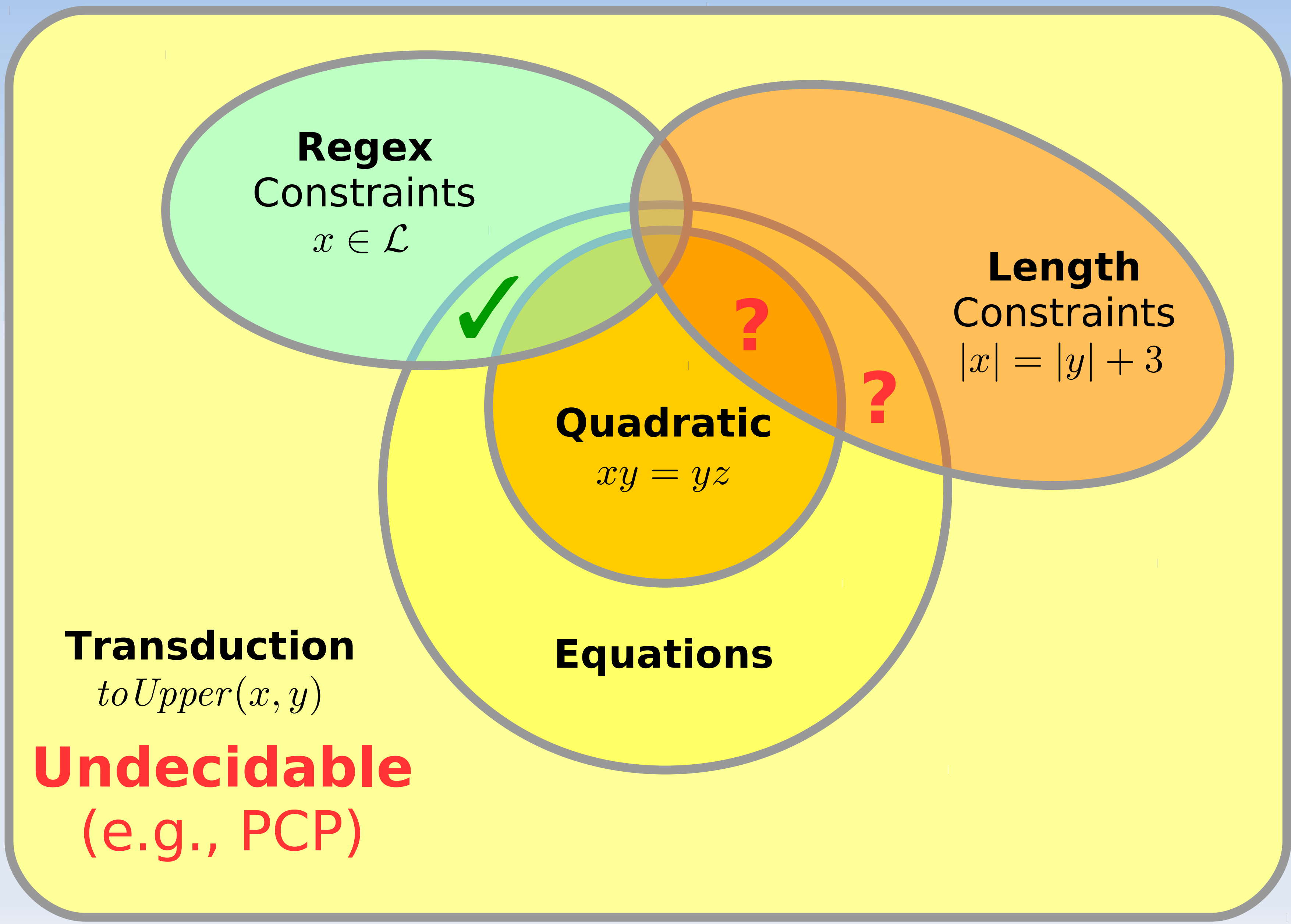
Combinations ...



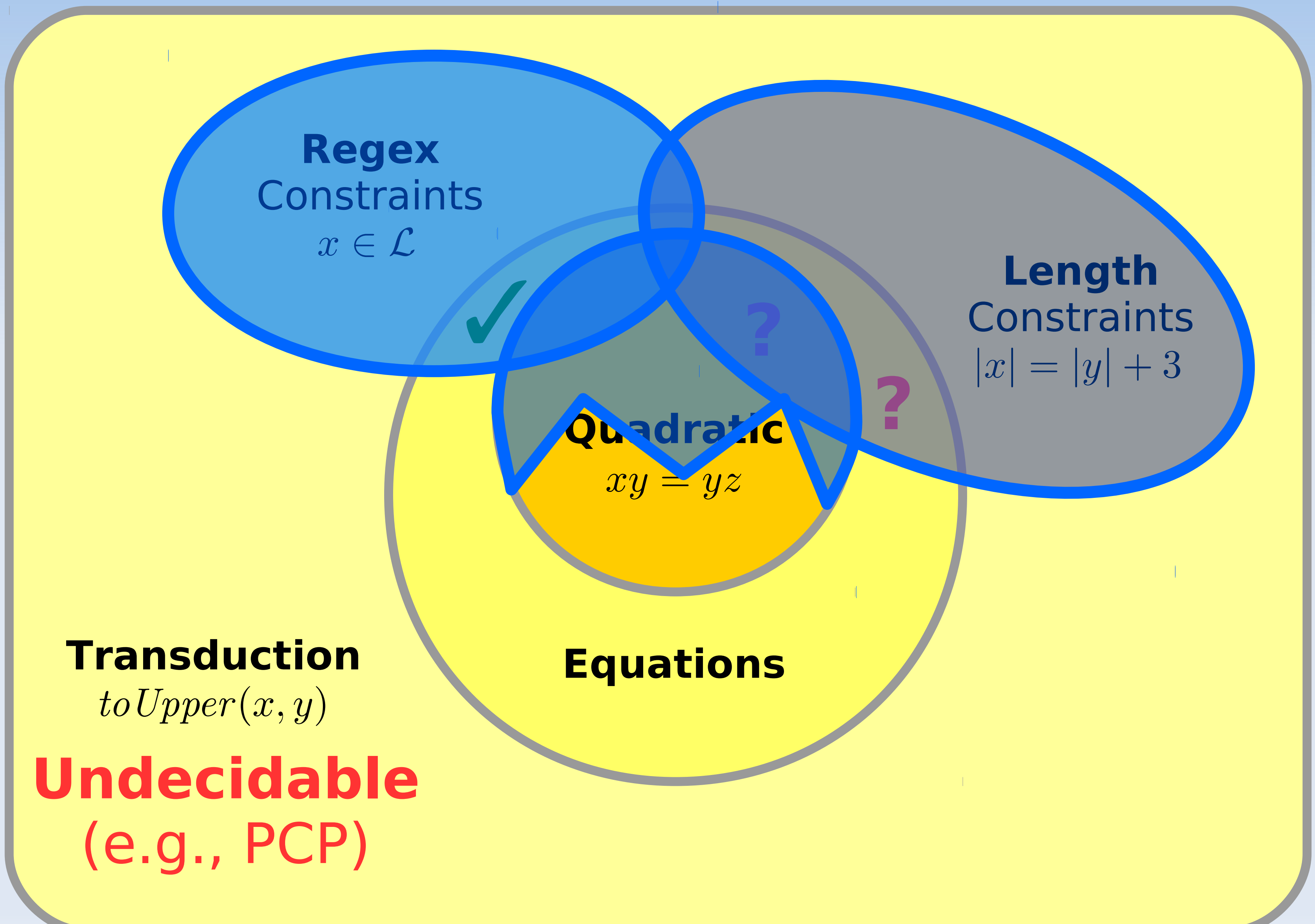
Combinations ...



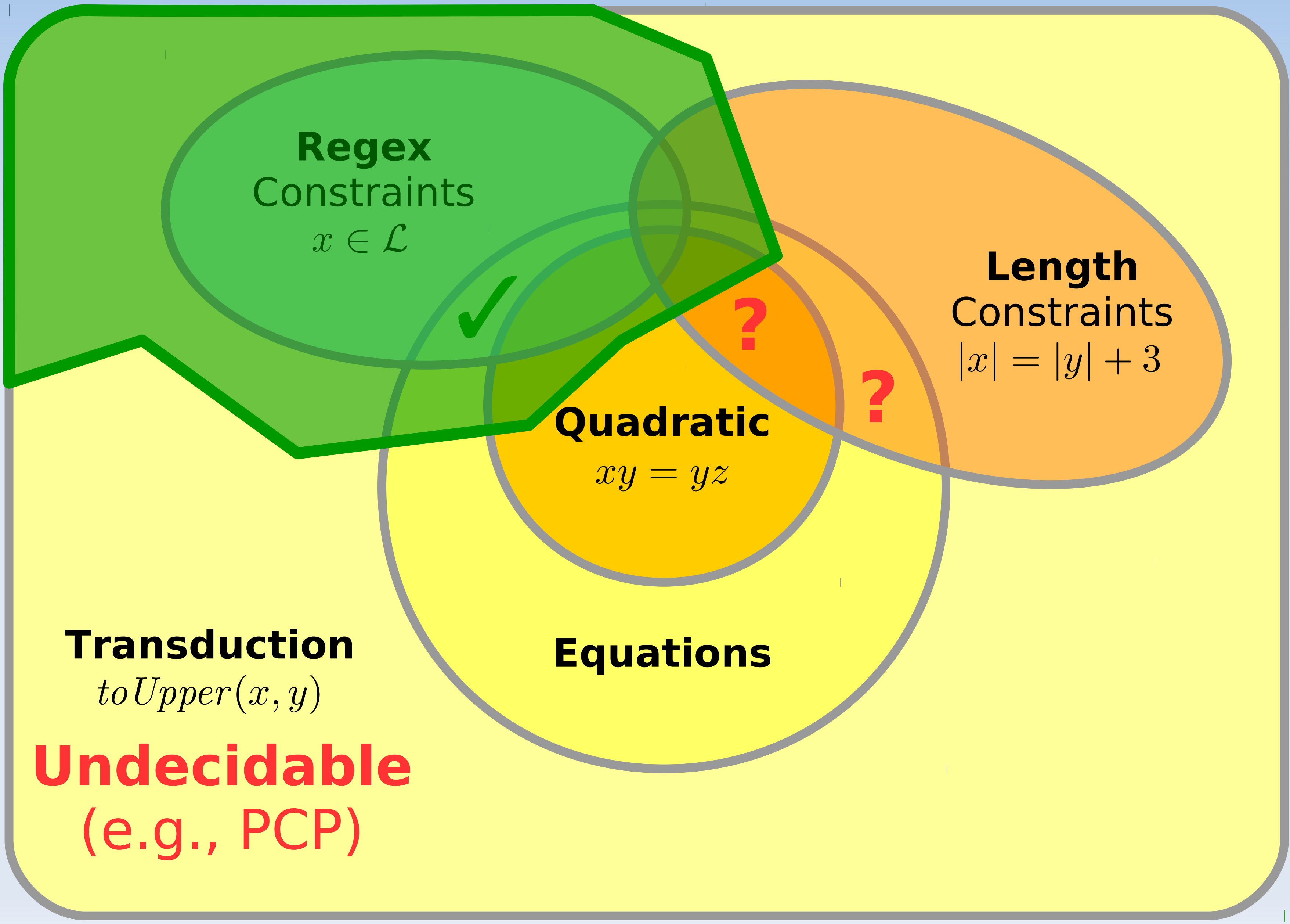
Combinations ...



Combinations ...



Combinations ...



The Norn fragment

1. Boolean structure
2. Acyclic (linear) word equations
3. Regex memberships
4. Length constraints

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, Jari Stenman: String Constraints for Verification. CAV 2014

The Norn fragment

1. Boolean structure
2. Acyclic (linear) word equations
3. Regex memberships
4. Length constraints

(a decidable fragment)

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, Jari Stenman: String Constraints for Verification. CAV 2014

The Norn fragment

1. Boolean structure
2. Acyclic (linear) word equations
3. Regex memberships
4. Length constraints

Order in which procedure handles operators

(a decidable fragment)

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, Jari Stenman: String Constraints for Verification. CAV 2014

Examples

```
// Pre = (true)
String s= '';
// P1 = (s ∈ ϵ)
while(*){
    // P2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
    s= 'a' + s + 'b';
}
// P3 = P2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P3
```

1. Boolean structure

- Use standard DPLL/CDCL → Easy
- Just consider conjunctions of literals

- But we need to handle negation!
  - Negated word equations
  - Negated regex constraints
  - Negated length constraints

1. Boolean structure

- Use standard DPLL/CDCL → Easy
- Just consider conjunctions of literals

- But we need to handle negation!
  - Negated word equations ?
  - Negated regex constraints ✓
  - Negated length constraints ✓

1b. Negative word eqs.

Can be reduced to positive equations:

Lemma

$$x \neq y \quad \Leftrightarrow \quad \begin{cases} \exists a \in \Sigma, u \in \Sigma^*.\ x = y \cdot a \cdot u;\ \text{or} \\ \exists a \in \Sigma, u \in \Sigma^*.\ y = a \cdot a \cdot u;\ \text{or} \\ \exists a \neq b \in \Sigma, p, u, v \in \Sigma^*. \\ \qquad x = p \cdot a \cdot u \wedge y = p \cdot b \cdot v \end{cases}$$

1b. Negative word eqs.

Can be reduced to positive equations:

Lemma

$$x \neq y \quad \Leftrightarrow \quad \begin{cases} \exists a \in \Sigma, u \in \Sigma^*.\ x = y \cdot a \cdot u;\ \text{or} \\ \exists a \in \Sigma, u \in \Sigma^*.\ y = a \cdot a \cdot u;\ \text{or} \\ \exists a \neq b \in \Sigma, p, u, v \in \Sigma^*. \\ \qquad x = p \cdot a \cdot u \wedge y = p \cdot b \cdot v \end{cases}$$

Large alphabets
→ *a*, *b* need to be
handled symbolically
in practice

1b. Negative word eqs.

Can be reduced to positive equations:

Lemma

$$x \neq y \quad \Leftrightarrow \quad \begin{cases} \exists a \in \Sigma, u \in \Sigma^*.\ x = y \cdot a \cdot u; \text{ or} \\ \exists a \in \Sigma, u \in \Sigma^*.\ y = a \cdot a \cdot u; \text{ or} \\ \exists a \neq b \in \Sigma, p, u, v \in \Sigma^*. \\ \qquad x = p \cdot a \cdot u \wedge y = p \cdot b \cdot v \end{cases}$$

Theorem

Any Boolean combination of word equations can be reduced to a single word equation with the same set of solutions (when projected to the original set of variables).

2. Acyclic word equations

- Reduce to solved form by systematic application of Nielsen's transformation:

$$x_1 = t_1 \wedge \cdots \wedge x_n = t_n$$

($x_1, \ldots, x_n$ do not occur in $t_1, \ldots, t_n$)

- After that, eliminate equations by inlining!

3. Regular expressions

- Membership tests with **concatenation** can be split:

$$s \cdot t \in \mathcal{L} \quad \rightsquigarrow \quad \bigvee_{i=1}^{n} s \in \mathcal{L}_1^i \wedge t \in \mathcal{L}_2^i$$

- Tests with **same left-hand side** can be merged:

$$x \in \mathcal{L}_1 \wedge x \in \mathcal{L}_2 \quad \rightsquigarrow \quad x \in \mathcal{L}_1 \cap \mathcal{L}_2$$

3. Regular expressions

- Membership tests with **concatenation** can be split:

$$s \cdot t \in \mathcal{L} \quad \rightsquigarrow \quad \bigvee_{i=1}^{n} s \in \mathcal{L}_1^i \wedge t \in \mathcal{L}_2^i$$

Disjunction over states of automaton representing $\mathcal{L}$

- Tests with **same left-h** be merged:

$$x \in \mathcal{L}_1 \wedge x \in \mathcal{L}_2 \quad \rightsquigarrow \quad x \in \mathcal{L}_1 \cap \mathcal{L}_2$$

4. Length constraints

- Compute the **length abstraction** of each regex constraint:

$$x \in \mathcal{L} \quad \rightsquigarrow \quad |x| \in \{|w| \mid w \in \mathcal{L}\}$$

- Conjoin length abstractions with other length constraints and check **satisfiability**

4. Length constraints

- Compute the **length abstraction** of each regex constraint:

$$x \in \mathcal{L} \quad \rightsquigarrow \quad |x| \in \{|w| \mid w \in \mathcal{L}\}$$

- Conjoin length abstractions with other length constraints and check **satisfiability**

A Presburger formula that can be extracted in linear time from $\mathcal{L}$

5. Optimisations ...

- E.g., exploit length information when splitting equations or regexes

(still too slow ...)

The Sloth fragments

1. Boolean structure (no negation)
2. Straight-line word equations
3. *n*-track transducer constraints

Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, Tomás Vojnar: String constraints with concatenation and transducers solved efficiently. PACMPL 2(POPL): 4:1-4:32 (2018)

The Sloth fragments

1. Boolean structure (no negation)
2. Straight-line word equations
3. *n*-track transducer constraints

→ also decidable!

Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, Tomás Vojnar: String constraints with concatenation and transducers solved efficiently. PACMPL 2(POPL): 4:1-4:32 (2018)

Example

Conclusions:
Are we there yet?

Expressiveness

Efficiency

Precision/
guarantees

Joint work with ...

- Parosh Aziz Abdulla
- Mohamed Faouzi Atig
- Yu-Fang Chen
- Bui Phi Diep
- Lukás Holík
- Petr Janků
- Anthony W. Lin
- Ahmed Rezine
- Jari Stenman
- Tomás Vojnar
- *and others*

Further topics

- The SMT-LIB standard for strings (work in progress ...)
- Solver applying under- and over-approximations
- Context-free grammars
- Model counting
- ...