Solving String Constraints via Regular Constraint Propagation

MOSCA 2025

Oliver Markgraf, Matthew Hague, Artur Jeż, Anthony Widjaja Lin and Philipp Rümmer

Zagreb, July 22, 2025

Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Why Study Regular Constraint Propagation?

Regular Constraint Propagation is:

- Simple — it just propagates regular constraints through function applications
- Widely used — many solvers use it already in limited capacity

But how far can it really take us?

Why Study Regular Constraint Propagation?

Regular Constraint Propagation is:

- Simple — it just propagates regular constraints through function applications
- Widely used — many solvers use it already in limited capacity

But how far can it really take us?
Our hopes:

- Maybe it's more powerful than expected — even complete for some interesting fragment?
- Maybe it's a strategy every solver could benefit from?

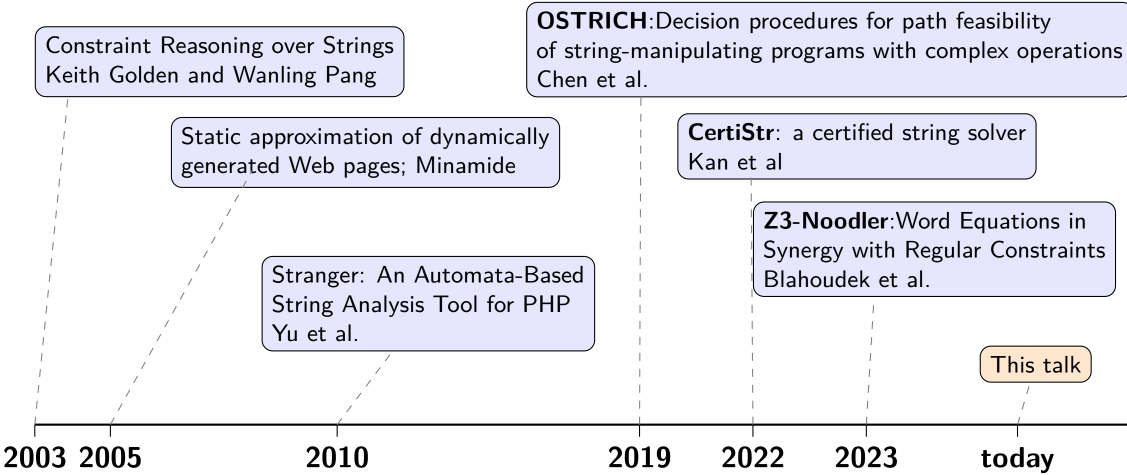
Takeaway

Regular Constraint Propagation alone is surprisingly powerful: complete for a large decidable fragment, simple yet effective in practice.

Agenda

1. Regular Constraint Propagation
2. Orderable Fragment
3. Experimental Results

History of Constraint Propagation in String Solving



RPTU

Normalize Decimal Function

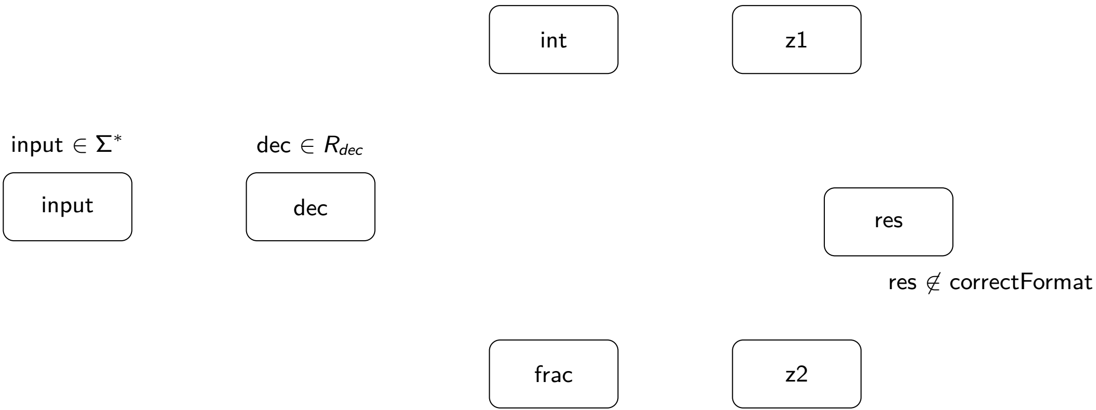
```
function normalize(decimal) {
        decimal = decimal.trim();
        const decimalReg = /^(\d+)\.?( \d*)$/;
        var decomp = decimal.match(decimalReg);
        var result = "";
        if (decomp) {
                var integer = decomp[1].replace(/^0+/, "");
                var fractional = decomp[2].replace(/0+$/, "");
                if (integer !== "") result = integer; else result = "0";
                if (fractional !== "") result = result + "." + fractional;
        }
        return result;
}
```

Normalize a decimal by trimming whitespace and removing leading and trailing zeros.

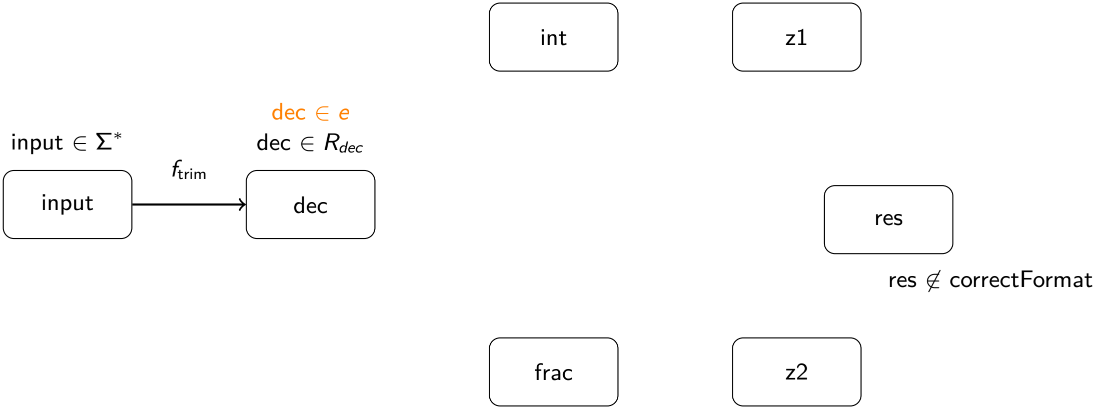
From Code to Constraint

`decimal = decimal.trim()`	$\longrightarrow$	$dec = f_{\text{trim}}(input)$
`decimal.match(decimalReg)`	$\longrightarrow$	$dec \in R_{\text{dec}} \wedge dec = f_{\text{concat}}(int, "\text{.}", frac)$
`integer = decomp[1].replace(/^0+/), and` `fractional = decomp[2].replace(/0+$/)`	$\longrightarrow$	$z_1 = f_{\text{lstrip0}}(int)$ $\wedge z_2 = f_{\text{rstrip0}}(frac)$ $\wedge res = f_{\text{concat}}(z_1, "\text{.}", z_2)$
`Verification condition`	$\longrightarrow$	$res \notin \texttt{correctFormat}$

RCP Constraint Graph

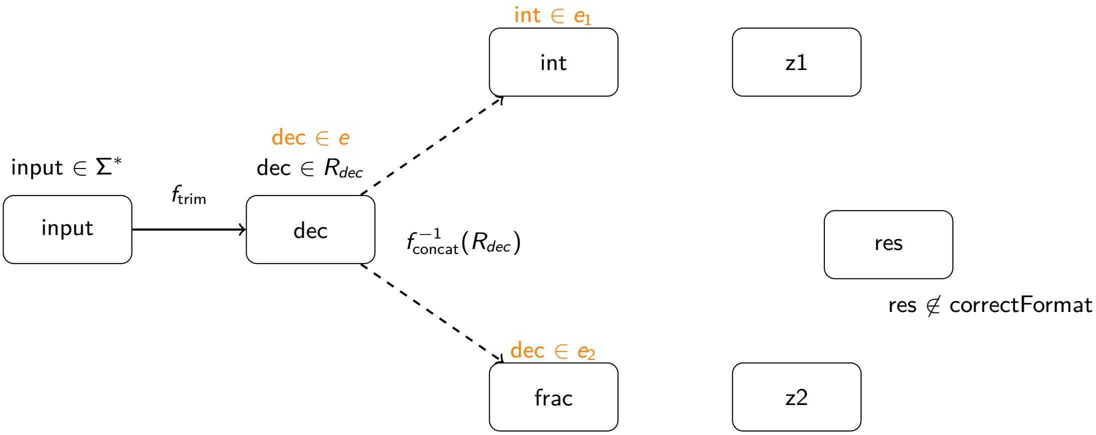


RCP Constraint Graph



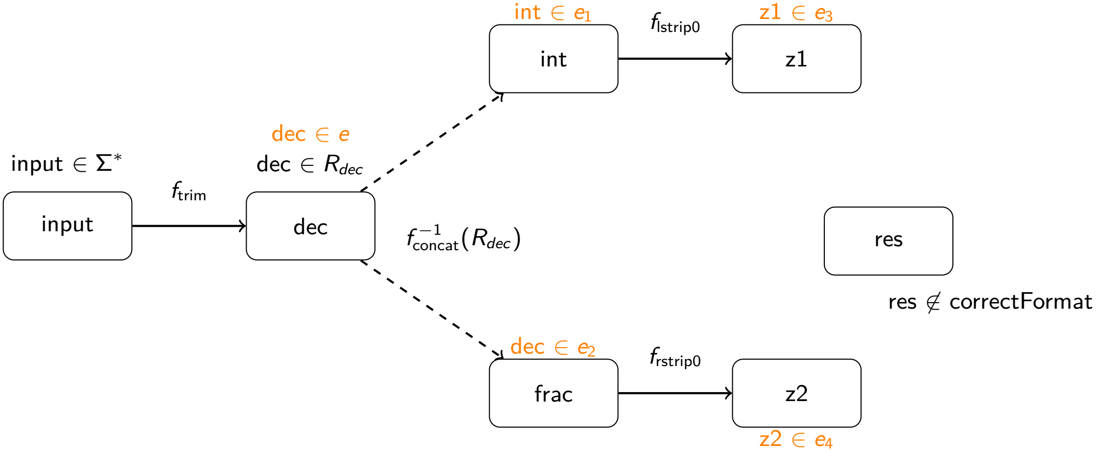
RPTU

RCP Constraint Graph

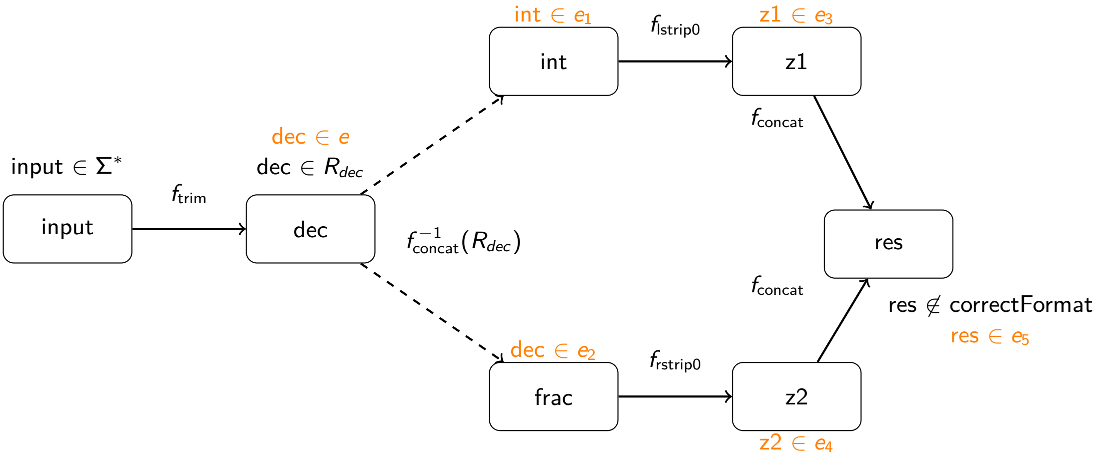


RPTU

RCP Constraint Graph



RCP Constraint Graph



1 Regular Constraint Propagation

Normal Form of Formulas

$$\psi ::= \phi \mid \psi \wedge \psi \qquad\qquad \phi ::= x \in e \mid x = f(x_1, \ldots, x_n)$$

- Variables: x, $x_1, \ldots, x_n$
- Functions: f (e.g., concat, replaceAll, etc.)
- Regular expressions: e

Normal Form of Formulas

$$\psi \ ::= \ \phi \mid \psi \wedge \psi \qquad\qquad \phi \ ::= \ x \in e \mid x = f(x_1, \ldots, x_n)$$

- Variables: x, $x_1, \ldots, x_n$
- Functions: f (e.g., concat, replaceAll, etc.)
- Regular expressions: e

Examples:

Formula	Description	In normal form?
$x \in (a \mid b)^*$	Regular constraint	✓

Normal Form of Formulas

$$\psi ::= \phi \mid \psi \wedge \psi \qquad \phi ::= x \in e \mid x = f(x_1, \ldots, x_n)$$

- Variables: x, $x_1, \ldots, x_n$
- Functions: f (e.g., concat, replaceAll, etc.)
- Regular expressions: e

Examples:

Formula	Description	In normal form?
$x \in (a\|b)^*$	Regular constraint	✓
$x = \texttt{replaceAll}(y, a, bb) \wedge x \in A$	Conjunction of allowed forms	✓

Normal Form of Formulas

$$\psi ::= \phi \mid \psi \wedge \psi \qquad\qquad \phi ::= x \in e \mid x = f(x_1, \ldots, x_n)$$

- Variables: x, $x_1, \ldots, x_n$
- Functions: f (e.g., concat, replaceAll, etc.)
- Regular expressions: e

Examples:

Formula	Description	In normal form?
$x \in (a\|b)^*$	Regular constraint	✓
$x = \texttt{replaceAll}(y, a, bb) \wedge x \in A$	Conjunction of allowed forms	✓
$x = \texttt{f}(g(y))$	Nested application	✗

Normal Form of Formulas

$$\psi \ ::= \ \phi \mid \psi \land \psi \qquad\qquad \phi \ ::= \ x \in e \mid x = f(x_1, \ldots, x_n)$$

- Variables: x, $x_1, \ldots, x_n$
- Functions: f (e.g., concat, replaceAll, etc.)
- Regular expressions: e

Examples:

Formula	Description	In normal form?
$x \in (a\|b)^*$	Regular constraint	✓
$x = \texttt{replaceAll}(y, a, bb) \land x \in A$	Conjunction of allowed forms	✓
$x = \texttt{f}(g(y))$	Nested application	✗
$x = \texttt{f}(z) \land z = \mathrm{g}(y)$	Resolved nesting	✓

What is a Proof System?

Gentzen-style sequent rules:

$$\frac{\text{Premise 1} \quad \text{Premise 2}}{\text{Conclusion}}\,\text{label}$$

- We read **bottom-up**: from conclusion to premises
- A line means a deduction step
- Commas stand for conjunctions: $\texttt{x} \in \texttt{A}, \texttt{x = f(...)} = \texttt{x} \in \texttt{A} \wedge \texttt{x = f(...)}$
- Branching occurs in rules for disjunction or nondeterminism
- The proof succeeds if all branches are closed by an Axiom

Forward Propagation (Fwd-Prop)

Concrete Example:

- $x_1 \in (a|b)^*$
- $x_2 \in b^*$
- $x = f_{concat}(x_1, x_2)$

Therefore:

$$\mathbf{x} \in (\mathbf{a}|\mathbf{b})^* \mathbf{b}^*$$

Instantiated Proof Step:

$$\frac{\mathbf{x} \in (\mathbf{a}|\mathbf{b})^* \mathbf{b}^*,\ x = f_{concat}(x_1, x_2),\ x_1 \in (a|b)^*,\ x_2 \in b^*}{x = f_{concat}(x_1, x_2),\ x_1 \in (a|b)^*,\ x_2 \in b^*}\ \texttt{[Fwd-Prop]}$$

Forward Propagation (Fwd-Prop)

Concrete Example:

- $x_1 \in (a|b)^*$
- $x_2 \in b^*$
- $x = f_{concat}(x_1, x_2)$

Therefore:

$$\mathbf{x} \in (\mathbf{a}|\mathbf{b})^*\mathbf{b}^*$$

Instantiated Proof Step:

$$\frac{\mathbf{x} \in (\mathbf{a}|\mathbf{b})^*\mathbf{b}^*,\ x = f_{concat}(x_1, x_2),\ x_1 \in (a|b)^*,\ x_2 \in b^*}{x = f_{concat}(x_1, x_2),\ x_1 \in (a|b)^*,\ x_2 \in b^*}\ \texttt{[Fwd-Prop]}$$

General Rule:

$$\frac{\Gamma, \mathbf{x} \in \mathbf{e}, x = f(x_1, \ldots, x_n), x_1 \in e_1, \ldots, x_n \in e_n}{\Gamma,\ x = f(x_1, \ldots, x_n),\ x_1 \in e_1,\ \ldots,\ x_n \in e_n}\ \texttt{[Fwd-Prop]}$$

where $L(e) = f(L(e_1), \ldots, L(e_n))$

Forwardable Functions

Definition (Forwardable)

A function $f : (\Sigma^*)^k \to \Sigma^*$ is *forwardable* if, for all regular languages $L_1, \ldots, L_k \subseteq \Sigma^*$, the image

$$f(L_1, \ldots, L_k) \quad \text{is regular},$$

and there is an algorithm to compute a representation of this language from representations of $L_1, \ldots, L_k$.

Forwardable Functions

Definition (Forwardable)

A function $f : (\Sigma^*)^k \to \Sigma^*$ is *forwardable* if, for all regular languages $L_1, \ldots, L_k \subseteq \Sigma^*$, the image

$$f(L_1, \ldots, L_k) \quad \text{is regular,}$$

and there is an algorithm to compute a representation of this language from representations of $L_1, \ldots, L_k$.

✓ Forwardable Examples

- `concat` — if arguments used linearly (e.g., $x = yz$)
- Rational transductions
- `replaceAll`(input, const, const)

✗ Not Forwardable

- `concat` — if a variable is reused (e.g., $x = yy$)
- `replaceAll`(input, const, x) — variable replacement

When Forward Propagation Fails

Example: Non-forwardable function — self-concatenation

Let:

$$x = f_{concat}(y, y) \qquad \text{and} \qquad y \in \{a, b\}^*$$

When Forward Propagation Fails

Example: Non-forwardable function — self-concatenation

Let:

$$x = f_{concat}(y, y) \qquad \text{and} \qquad y \in \{a, b\}^*$$

Then:

$$x \in \{ww \mid w \in \{a, b\}^*\} \quad \text{(the set of all string self-concatenations)}$$

RPTU

When Forward Propagation Fails

Example: Non-forwardable function — self-concatenation

Let:

$$x = f_{concat}(y, y) \qquad \text{and} \qquad y \in \{a, b\}^*$$

Then:

$$x \in \{ww \mid w \in \{a, b\}^*\} \quad \text{(the set of all string self-concatenations)}$$

This language is **not regular**.

- So we cannot compute an exact regular constraint on x from y via forward propagation
- Any forward propagation (e.g. $x \in (a|b)^*(a|b)^*$) is an over-approximation

RPTU

Backward Propagation (Bwd-Prop)

Example:

- $x = f_{concat}(y, y)$
- $x \in a^* b$

Then backward propagation gives three branches:

$$f_{concat}^{-1}(a^* b) = \begin{cases} (1) \quad y \in \varepsilon, \quad y \in a^* b \\ (2) \quad y \in a^*, \quad y \in b \\ (3) \quad y \in a^* b, \quad y \in \varepsilon \end{cases}$$

Backward Propagation (Bwd-Prop)

Example:

- $x = f_{concat}(y, y)$
- $x \in a^* b$

Then backward propagation gives three branches:

$$f_{concat}^{-1}(a^* b) = \begin{cases} (1) \quad y \in \varepsilon, \quad y \in a^* b \\ (2) \quad y \in a^*, \quad y \in b \\ (3) \quad y \in a^* b, \quad y \in \varepsilon \end{cases}$$

$$\dfrac{\dfrac{}{x = f_{concat}(y, y),\ \mathbf{y} \in \mathbf{a}^*,\ \mathbf{y} \in \mathbf{b},\ x \in a^* b} \qquad \dfrac{}{x = f_{concat}(y, y),\ \mathbf{y} \in \mathbf{a}^*\mathbf{b},\ \mathbf{y} \in \varepsilon,\ x \in a^* b}}{x = f_{concat}(y, y),\ x \in a^* b} \ \texttt{[Bwd-Prop]}$$

General Rule: Backward Propagation (Bwd-Prop)

Let $f^{-1}(L(e)) = \bigcup_{i=1}^{k} L(e_1^i) \times \cdots \times L(e_n^i)$

Then we apply the rule:

$$x \in e, \quad x = f(x_1, \ldots, x_n) \quad \rightsquigarrow \quad \text{branches over } x_j \in e_j^i$$

RPTU

General Rule: Backward Propagation (Bwd-Prop)

Let $f^{-1}(L(e)) = \bigcup_{i=1}^{k} L(e_1^i) \times \cdots \times L(e_n^i)$

Then we apply the rule:

$$x \in e, \quad x = f(x_1, \ldots, x_n) \quad \rightsquigarrow \quad \text{branches over } x_j \in e_j^i$$

$$\frac{\Gamma,\ x = f(x_1,\ldots,x_n),\ x \in e,\ x_1 \in e_1^1,\ldots,x_n \in e_n^1 \qquad \ldots \qquad \Gamma,\ x = f(x_1,\ldots,x_n),\ x \in e,\ x_1 \in e_1^k,\ldots,x_n \in e_n^k}{\Gamma,\ x = f(x_1,\ldots,x_n),\ x \in e}\ \texttt{[Bwd-Prop]}$$

Backwardable Functions

Definition (Backwardable)

A function $f : (\Sigma^*)^k \to \Sigma^*$ is *backwardable* if, for all regular languages $L \subseteq \Sigma^*$:

- The preimage $f^{-1}(L)$ is a **recognizable relation**, and
- There exists an algorithm to compute a representation of $f^{-1}(L)$ from a representation of L

Recognizable relation:

$$R \subseteq (\Sigma^*)^k \text{ is recognizable if } R = \bigcup_{i=1}^{m} L_{i,1} \times \cdots \times L_{i,k}, \text{ with each } L_{i,j} \text{ regular}$$

This is exactly the structure used in the `[Bwd-Prop]` rule. The representation is not unique.

Backwardable Functions

✓ Backwardable

- `concat` — even with duplicated variables (e.g., $x = yy$)
- Rational transductions
- `replaceAll`(x, constant, variable)
- Regular replacement with capture groups
- Streaming string transducers
- Poly-regular functions

✗ Not Backwardable

- `replaceAll`(x, variable, variable)

Question: How do we end a proof?

A branch closes when we derive an unsatisfiable constraint, i.e. a variable constrained to the empty language:

$$\frac{}{\Gamma,\ x \in e_1,\ \ldots,\ x \in e_n}\ \texttt{[Close]}$$

where $L(e_1) \cap \cdots \cap L(e_n) = \emptyset$

Question: How do we end a proof?

Both branches close

$$f_{concat}^{-1}(a^* b) = \begin{cases} (1) \quad y \in a^*, \quad y \in b \quad \Rightarrow \quad L = \emptyset \\ (2) \quad y \in \varepsilon, \quad y \in a^* b \quad \Rightarrow \quad L = \emptyset \end{cases}$$

Question: How do we end a proof?

Both branches close

$$f_{concat}^{-1}(a^*b) = \begin{cases} (1) & y \in a^*, \quad y \in b \quad \Rightarrow \quad L = \emptyset \\ (2) & y \in \varepsilon, \quad y \in a^*b \quad \Rightarrow \quad L = \emptyset \end{cases}$$

$$\dfrac{\dfrac{}{x = f_{concat}(y, y),\ \mathbf{y} \in \mathbf{a}^*,\ \mathbf{y} \in \mathbf{b},\ x \in a^*b}\,[\texttt{Close}] \qquad \dfrac{}{x = f_{concat}(y, y),\ \mathbf{y} \in \mathbf{a}^*\mathbf{b},\ \mathbf{y} \in \varepsilon,\ x \in a^*b}\,[\texttt{Close}]}{x = f_{concat}(y, y),\ x \in a^*b}\,[\texttt{Bwd-Prop}]$$

The Intersect Rule: Why We Need It

Purpose: Combine multiple constraints on a variable into a single regular expression.

Rule:

$$\texttt{[Intersect]} \qquad \Gamma, x \in e_1, \ldots, x \in e_n \quad \rightsquigarrow \quad \Gamma, x \in e \quad \text{where } L(e) = \bigcap_i L(e_i)$$

Why it is needed:

$$f^{-1}(L_1 \cap L_2) = f^{-1}(L_1) \cap f^{-1}(L_2) \quad \text{(OK for backward)}$$

$$f(L_1 \cap L_2) \subseteq f(L_1) \cap f(L_2) \quad \text{(NOT equal in general)}$$

Counterexample:
Let:

$$A_1 = \{a,\ aa\}, \quad A_2 = \{aa,\ aaa\}, \quad B = \{aab,\ b\}$$

Then: $aaab \in (A_1 \cdot B) \cap (A_2 \cdot B)$, but $aaab \notin ((A_1 \cap A_2) \cdot B)$

So we intersect first and then apply forward propagation.

2 Orderable Fragment

Orderable Fragment in Context

Fragment	Rough Idea	Captured by Orderable?
Straight-line	Acyclic use of functions, define variables at most once	✓Yes
Chain-free	Allows chaining, but restricts dependencies between variables	✓Yes
Orderable (new)	Captures both fragments via a simple propagation order	(*) This talk

RPTU

Straight-Line Fragment (SL)

Structure:

- Each variable defined at most once
- Definitions follow a linear order — no reuse or cycles

Example:

$$
\begin{aligned}
x_1 &= f_{concat}(x, x) \\
x_2 &= f_{concat}(x_1, x_1) \\
x_3 &= f_{concat}(x_1, x_2) \\
&\dots
\end{aligned}
$$

Captured by orderable: ✓Yes — propagation follows definition order.

Taolue Chen, Matthew Hague, Anthony Lin, Philipp Rümmer, and Zhilin Wu.
"Decision procedures for path feasibility of string-manipulating programs with complex operations." POPL, 2019.

Chain-Free Fragment (CF)

Structure:

- Allows re-use of variables and chaining of constraints
- Use splitting graph to compute dependencies

Example:

$$
\begin{aligned}
x_1 &= f_{\text{concat}}(x, x) \\
x_2 &= f_{\text{concat}}(x_1, x_1) \\
x_3 &= f_{\text{concat}}(x_1, x_2) \\
x_3 &= f_{\text{concat}}(z_1, z_2) \qquad \textit{(second constraint on } x_3\textit{)}
\end{aligned}
$$

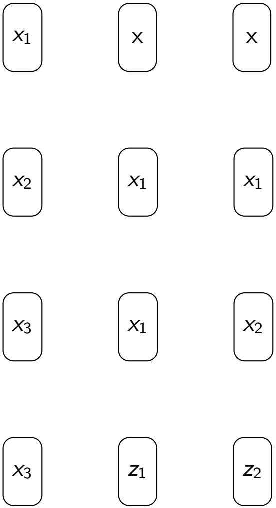
This is not straight-line (due to reuse), but still chain-free.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukáš Holík, and Petr Janků. 2019.
"Chain-Free String Constraints". ATVA, 2019

Chain-Free Example: Splitting Graph

Example:

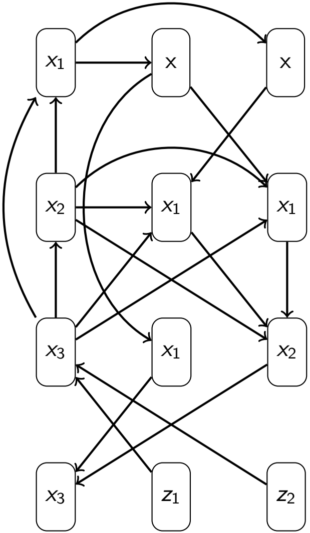
$$
\begin{aligned}
x_1 &= f_{\text{concat}}(x, x) \\
x_2 &= f_{\text{concat}}(x_1, x_1) \\
x_3 &= f_{\text{concat}}(x_1, x_2) \\
x_3 &= f_{\text{concat}}(z_1, z_2)
\end{aligned}
$$



Chain-Free Example: Splitting Graph

Example:

$$
\begin{aligned}
x_1 &= f_{\text{concat}}(x, x) \\
x_2 &= f_{\text{concat}}(x_1, x_1) \\
x_3 &= f_{\text{concat}}(x_1, x_2) \\
x_3 &= f_{\text{concat}}(z_1, z_2)
\end{aligned}
$$



Orderable Fragment

Structure:

- Allows re-use of variables and chaining of constraints
- Determine propagation order with the **Marking Algorithm**
- Visualize propagation order with **Flow Sequence** and **Flow Graph**

Example:

$$
\begin{aligned}
x_1 &= f_{\text{concat}}(x, x) \\
x_2 &= f_{\text{concat}}(x_1, x_1) \\
x_3 &= f_{\text{concat}}(x_1, x_2) \\
x_3 &= f_{\text{concat}}(z_1, z_2)
\end{aligned}
$$

Flow Direction:

Orderable Fragment

Structure:

- Allows re-use of variables and chaining of constraints
- Determine propagation order with the **Marking Algorithm**
- Visualize propagation order with **Flow Sequence** and **Flow Graph**

Example:

$$x_1 = f_{\text{concat}}(x, x)$$
$$x_2 = f_{\text{concat}}(x_1, x_1)$$
$$x_3 = f_{\text{concat}}(x_1, x_2)$$
$$x_3 = f_{\text{concat}}(z_1, z_2)$$

Flow Direction:

$$(x_3 = f_{\text{concat}}(z_1, z_2), \leftarrow)$$

Orderable Fragment

Structure:

- Allows re-use of variables and chaining of constraints
- Determine propagation order with the **Marking Algorithm**
- Visualize propagation order with **Flow Sequence** and **Flow Graph**

Example:

$$x_1 = f_{\text{concat}}(x, x)$$
$$x_2 = f_{\text{concat}}(x_1, x_1)$$
$$x_3 = f_{\text{concat}}(x_1, x_2)$$

Flow Direction:

$$(x_3 = f_{\text{concat}}(x_1, x_2), \rightarrow)$$
$$(x_3 = f_{\text{concat}}(z_1, z_2), \leftarrow)$$

Orderable Fragment

Structure:

- Allows re-use of variables and chaining of constraints
- Determine propagation order with the **Marking Algorithm**
- Visualize propagation order with **Flow Sequence** and **Flow Graph**

Example:

$$x_1 = f_{\text{concat}}(x, x)$$
$$x_2 = f_{\text{concat}}(x_1, x_1)$$

Flow Direction:

$$(x_2 = f_{\text{concat}}(x_1, x_1), \rightarrow)$$
$$(x_3 = f_{\text{concat}}(x_1, x_2), \rightarrow)$$
$$(x_3 = f_{\text{concat}}(z_1, z_2), \leftarrow)$$

Orderable Fragment

Structure:

- Allows re-use of variables and chaining of constraints
- Determine propagation order with the **Marking Algorithm**
- Visualize propagation order with **Flow Sequence** and **Flow Graph**

Example:

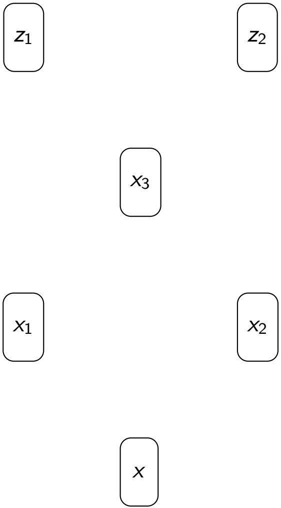
$$x_1 = f_{\text{concat}}(x, x)$$

Flow Direction:

$$(x_1 = f_{\text{concat}}(x, x), \rightarrow)$$
$$(x_2 = f_{\text{concat}}(x_1, x_1), \rightarrow)$$
$$(x_3 = f_{\text{concat}}(x_1, x_2), \rightarrow)$$
$$(x_3 = f_{\text{concat}}(z_1, z_2), \leftarrow)$$

Orderable Fragment

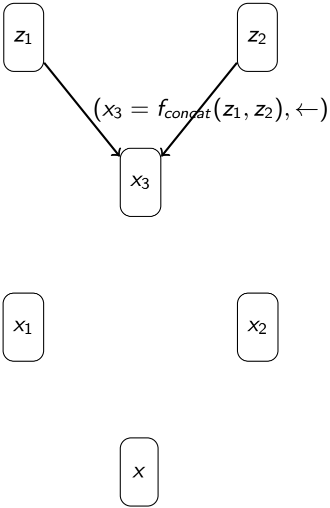
Example:



Orderable Fragment

Example:

$$(x_3 = f_{concat}(z_1, z_2), \leftarrow)$$

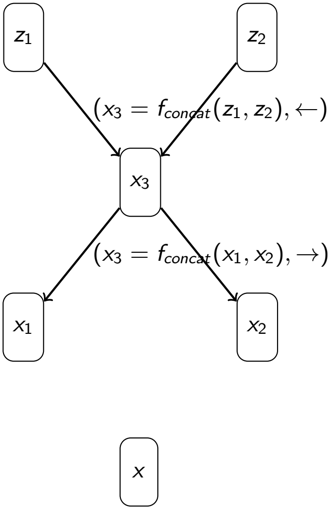


RPTU

Orderable Fragment

Example:

$$(x_3 = f_{concat}(z_1, z_2), \leftarrow)$$
$$(x_3 = f_{concat}(x_1, x_2), \rightarrow)$$

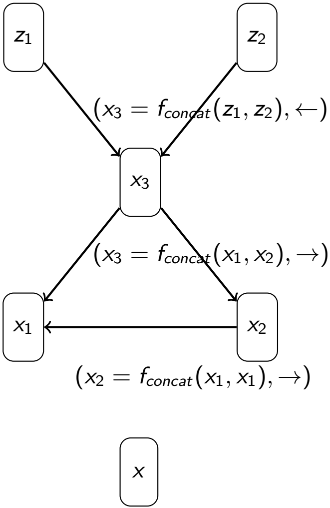


RPTU

Orderable Fragment

Example:

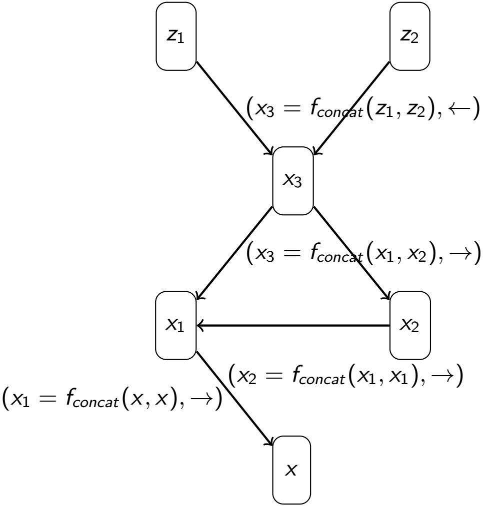
$$(x_3 = f_{concat}(z_1, z_2), \leftarrow)$$
$$(x_3 = f_{concat}(x_1, x_2), \rightarrow)$$
$$(x_2 = f_{concat}(x_1, x_1), \rightarrow)$$



Orderable Fragment

Example:

$$(x_3 = f_{concat}(z_1, z_2), \leftarrow)$$
$$(x_3 = f_{concat}(x_1, x_2), \rightarrow)$$
$$(x_2 = f_{concat}(x_1, x_1), \rightarrow)$$
$$(x_1 = f_{concat}(x, x), \rightarrow)$$



Main Result: Completeness of RCP

Theorem

Every unsatisfiable ***orderable*** *constraint admits a proof in the RCP system using only the following rules:*

$$\mathtt{Close},\quad \mathtt{Intersect},\quad \mathtt{Fwd\text{-}Prop},\quad \mathtt{Bwd\text{-}Prop}$$

Orderable Fragment — Is This the End for RCP?

RCP still succeeds beyond orderable:

$$x \in \texttt{"2"}^{+} \quad \wedge \quad y = \texttt{replaceAll}(x, \texttt{"2"}, \texttt{"10"})$$
$$z = \texttt{replaceAll}(x, \texttt{"2"}, \texttt{"01"}) \quad \wedge \quad y = z$$

- Not orderable — but RCP can still solve it
- The expressive power depends on **how regular expressions interact**

Benchmark Sets Overview

We evaluate solvers across three distinct benchmark sets:

- **SMT-LIB'24** (2000 benchmarks)
  - Derived from the latest SMT-LIB string division
  - Mix of real and synthetic formulas over regex, concat, replaceAll, etc.
- **PCP Benchmarks** (1000 instances)
  - Based on Post Correspondence Problem in form PCP[3,3]
  - Each instance has 3 dominos over binary alphabet, words of length 3
- **Bioinformatics Benchmarks** (1000 instances)
  - Models reverse transcription: RNA $\rightarrow$ DNA via `replaceAll`
  - Goal: find RNA y such that replacements yield a given DNA string, and y contains a specific motif

Research Questions

1. **Is Regular Constraint Propagation effective as a core solving technique?**
   - Can a propagation-based engine compete with modern solvers?
   - Does it scale to real-world and synthetic benchmarks?
2. **Is RCP complementary to existing techniques?**
   - Can it boost solvers when used as a component?
   - Does it solve instances others leave unanswered?

Experimental Evaluation

	SMT-LIB'24				PCP		Bioinformatics	
	Sat	Unsat	Unknown	Time(s)	Solved	Time(s)	Solved	Time(s)
RCP	1071	728	201	15416.8	901	7308.7	1000	5532.6
cvc5	1162	716	122	7954.3	0	60000.0	0	-
OSTRICH-BASE	833	653	514	32298.7	0	60000.0	1000	2596.2
OSTRICH-COMP	1009	733	258	22840.7	0	60000.0	1000	3911.0
Z3	1156	730	114	8286.0	0	60000.0	0	60000.0
Z3-alpha	1127	724	149	10681.2	0	60000.0	0	60000.0
Z3-Noodler	**1236**	**749**	**15**	797.0	0	60000.0	0	60000.0

Experimental Evaluation

Solver Combination	SMT-LIB'24 (2000)			PCP (1000)			Bioinformatics (1000)		
	U	C	I	U	C	I	U	C	I
RCP	201	–	–	99	–	–	0	–	–
cvc5 + RCP	61	61	0.50	99	901	0.90	0	1000	1.00
OSTRICH-COMP + RCP	68	197	0.74	99	901	0.90	0	0	0.00
Z3 + RCP	85	29	0.25	99	901	0.90	0	1000	1.00
Z3-alpha + RCP	87	65	0.43	99	901	0.90	0	1000	1.00
Z3-Noodler + RCP	**11**	4	0.27	99	901	0.90	0	1000	1.00

Conclusion and Future Work

Takeaways

- RCP is a simple and intuitive strategy
- It is complete for the **orderable fragment**, which subsumes known fragments
- As a core strategy, RCP is effective and competitive on diverse benchmarks
- **Every solver can benefit** from adopting RCP as a subroutine

Future Work

- Study power of RCP with other rules:
  - Power-introduction, cut rules, Nielsen splitting, Parikh reasoning
- Lift RCP to richer theories (e.g., sequences over infinite alphabets)
- Output and certify RCP proofs

Thank you

Oliver Markgraf, Matthew Hague, Artur Jeż, Anthony Widjaja Lin and Philipp Rümmer

markgraf(at)cs.uni-kl.de
RPTU in Kaiserslautern

Bonus: Additional Proof Rules for Satisfiability

[Solution]

$$\frac{}{x_1 \in e, \ \ldots, \ x_n \in e}$$

[Fwd-Elim]

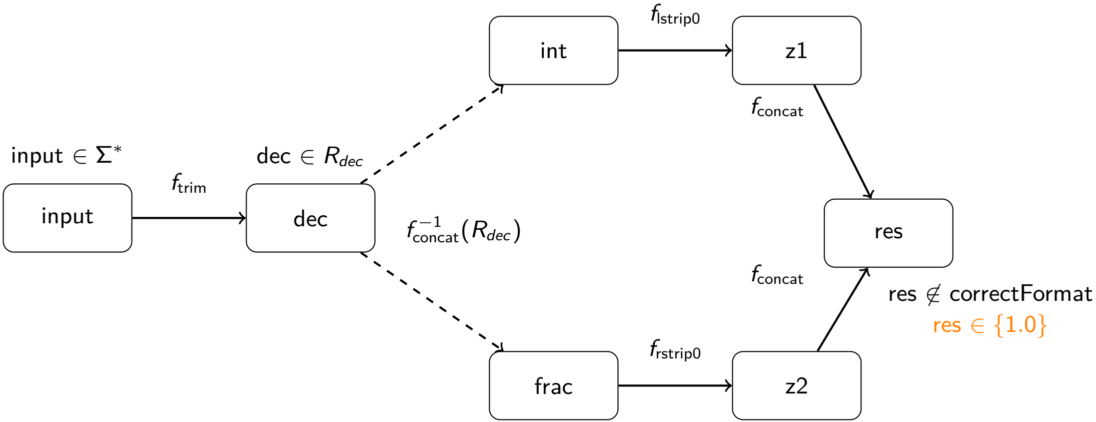
$$\frac{\Gamma, \mathbf{x} \in \mathbf{e}, x_1 \in e_1, \ldots, x_n \in e_n}{\Gamma, \ x = f(x_1, \ldots, x_n), \ x_1 \in e_1, \ \ldots, \ x_n \in e_n}$$

where $L(e) = f(L(e_1), \ldots, L(e_n))$ and $|L(e)| = 1$

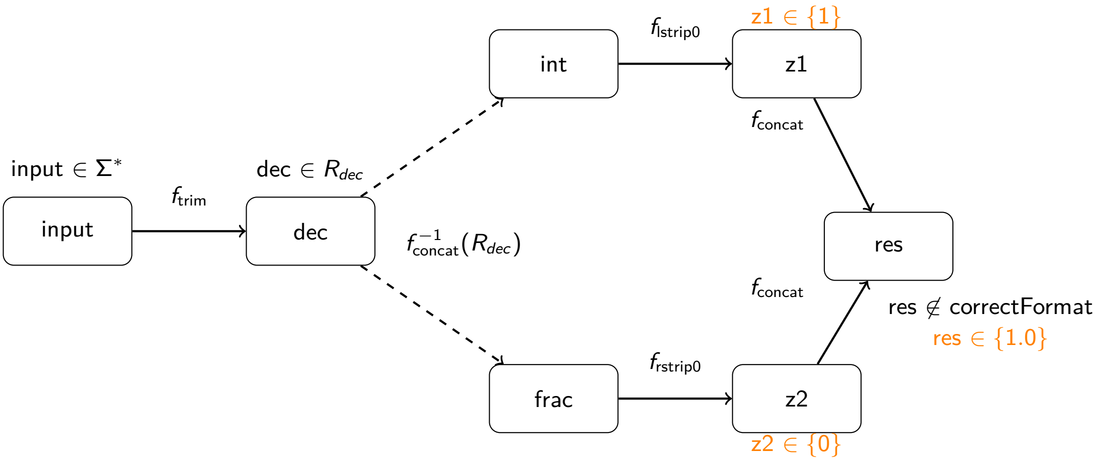
[Cut]

$$\frac{\Gamma, \ x \in e \qquad \Gamma, \ x \notin e}{\Gamma}$$

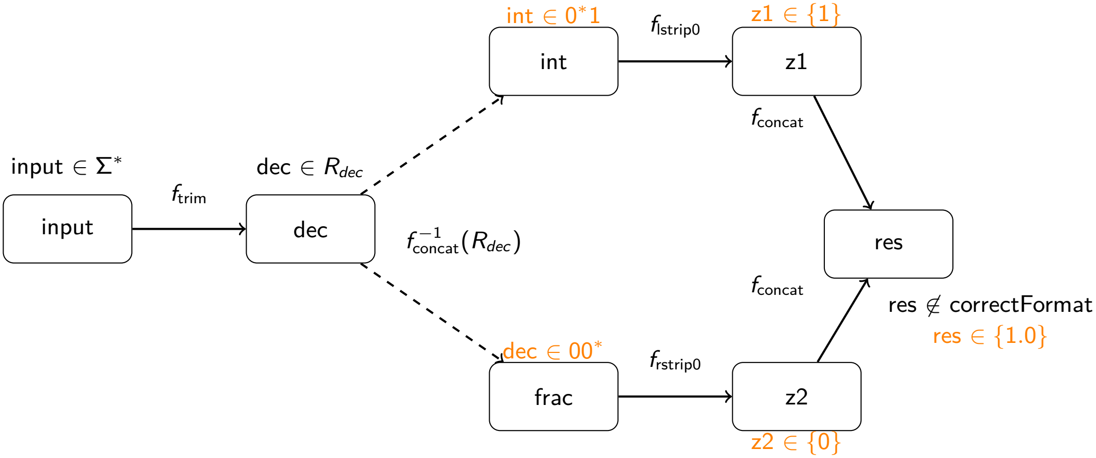
RCP Constraint Graph



RCP Constraint Graph

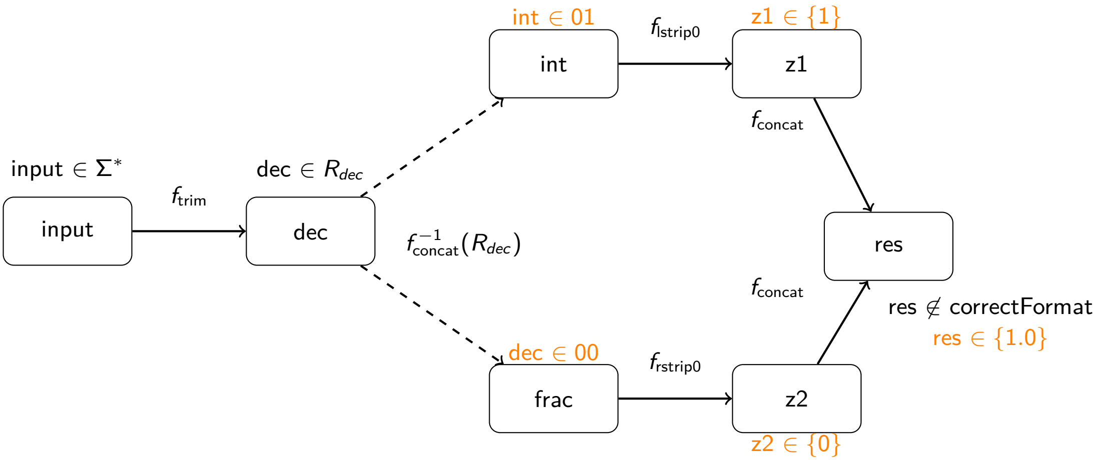


RCP Constraint Graph

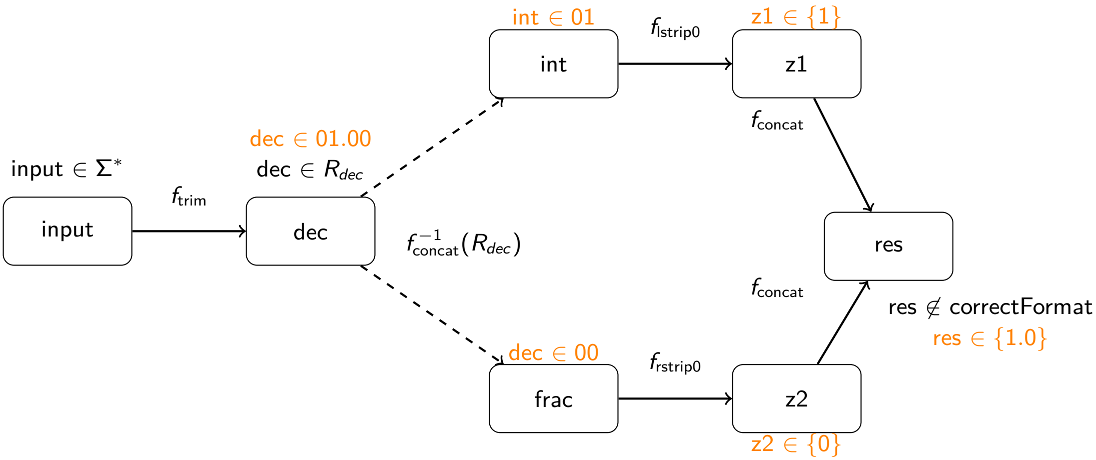


RPTU

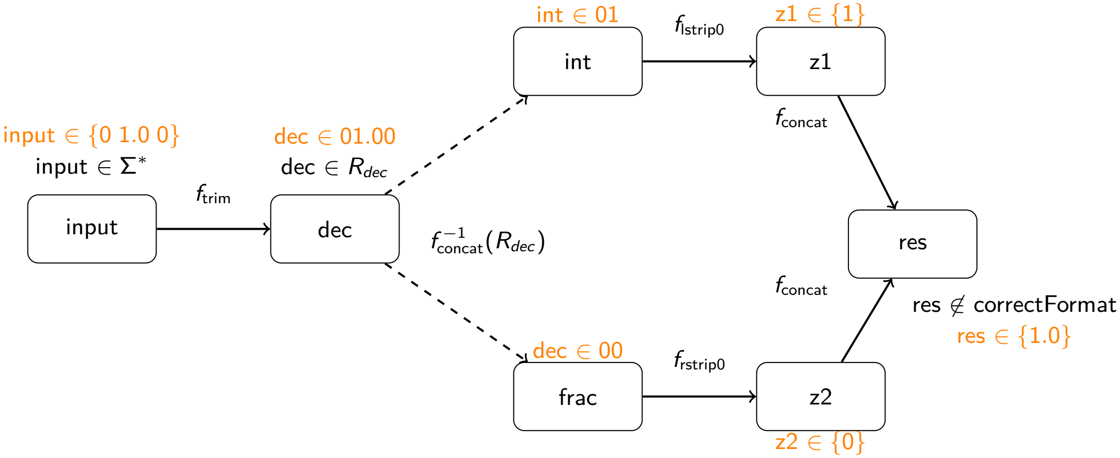
RCP Constraint Graph



RCP Constraint Graph



RCP Constraint Graph



Experimental Evaluation

Solver Combination	SMT-LIB'24 (2000)			PCP (1000)			Bioinformatics (1000)		
	U	C	I	U	C	I	U	C	I
RCP	201	–	–	99	–	–	0	–	–
cvc5 + RCP	61	61	0.50	99	901	0.90	0	1000	1.00
OSTRICH-COMP + RCP	68	197	0.74	99	901	0.90	0	0	0.00
Z3 + RCP	85	29	0.25	99	901	0.90	0	1000	1.00
Z3-alpha + RCP	87	65	0.43	99	901	0.90	0	1000	1.00
Z3-Noodler + RCP	**11**	4	0.27	99	901	0.90	0	1000	1.00
(cvc5 + OB) + RCP	61	2	0.03	99	901	0.90	0	0	0.00
(Z3 + OB) + RCP	85	5	0.06	99	901	0.90	0	0	0.00
(Z3-alpha + OB) + RCP	87	8	0.08	99	901	0.90	0	0	0.00
(Z3-Noodler + OB) + RCP	**4**	0	0.00	99	901	0.90	0	0	0.00