Regular Language Type Inference with Term Rewriting – ICFP 2020

Timothée Haudebourg
joined work with Thomas Genet and Thomas Jensen

October 23, 2020

Univ Rennes, Inria, IRISA – CELTIQUE Team

Introduction

For instance, we want to **automatically** solve this kind of safety problems:

```
# let rec sort cmp = function
    | [] -> []
    | x::l -> insert cmp x (sort cmp l) ;;
val insert : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list
val sort : ('a -> 'a -> bool) -> 'a list -> 'a list
# let rec sorted cmp = function
    | _ -> true
    | x::y::l -> cmp x y && sorted (y::l) ;;
val sorted : ('a -> 'a -> bool) -> 'a list -> bool
# forall l. sorted cmp (sort cmp l) ;;
```

Or goal is to answer the last query.

We do not deal with polymorphism (yet)

In this example instead, we define a very simple type equipped with a comparison function:

```
# type ab = A | B ;;
type ab = A | B

# let rec cmp x y = match x y with
    | B, A -> false
    | _, _ -> true
val cmp : ab -> ab -> bool
```

Our polymorphic signatures become:

```
val insert : (ab -> ab -> bool) -> ab -> ab list -> ab list
val sort : (ab -> ab -> bool) -> ab list -> ab list
val sorted : (ab -> ab -> bool) -> ab list -> bool
```

This is a type checking problem

Our safety problem becomes:

```
val insert : (ab -> ab -> bool) -> ab -> ab list -> ab list
val sort : (ab -> ab -> bool) -> ab list -> ab list
val sorted : (ab -> ab -> bool) -> ab list -> bool

# forall l. sorted cmp (sort cmp l) ;;
```

This is equivalent to the following refinement-type checking problem:

```
# function l -> ( sorted cmp (sort cmp l) : { true } ) ;;
```

Solving by refining the signature of the program

We need to infer precise types for `sorted` and `sort` to type the pattern:

```
( sorted cmp (sort cmp (l : ab list)) : { true } ) ;;
```

A possible solution is:

```
insert : { cmp } -> ab -> { sorted lists } -> { sorted lists }
sort : { cmp } -> ab list -> { sorted list }
sorted : { cmp } -> { sorted lists } -> { true }
```

This is a **regular abstraction** of the program:

- all the (refinement) types can be described with **regular tree languages**
- it captures all the **invariants** necessary to prove our property

Solving by refining the signature of the program

We need to infer precise types for `sorted` and `sort` to type the pattern:

```
( sorted cmp (sort cmp (l : ab list)) : { true } ) ;;
```

A possible solution is:

```
insert : { cmp } -> ab -> { sorted lists } -> { sorted lists }
sort : { cmp } -> ab list -> { sorted list }
sorted : { cmp } -> { sorted lists } -> { true }
```

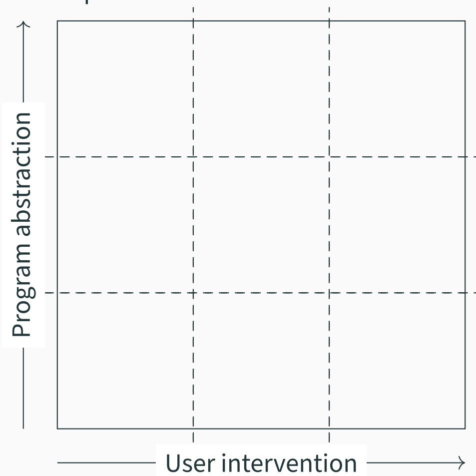
This is a **regular abstraction** of the program:

- all the (refinement) types can be described with **regular tree languages**
- it captures all the **invariants** necessary to prove our property

Our goal is to automatically infer such regular abstractions.

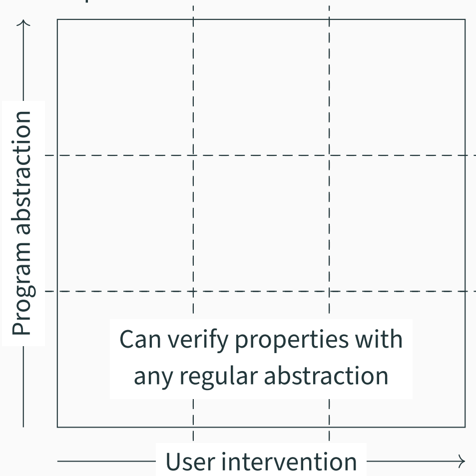
Current verification techniques vs. regular problems

Techniques classification:



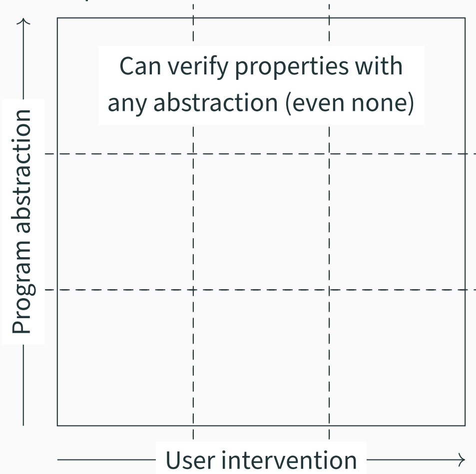
Current verification techniques vs. regular problems

Techniques classification:



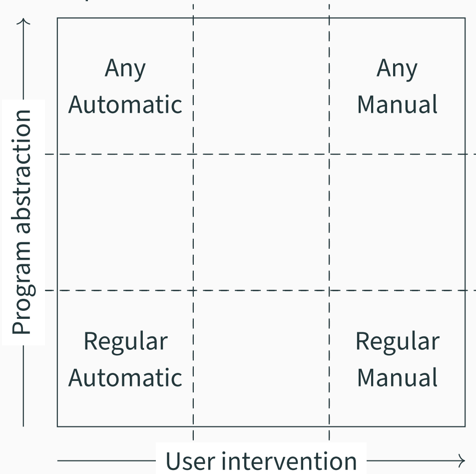
Current verification techniques vs. regular problems

Techniques classification:



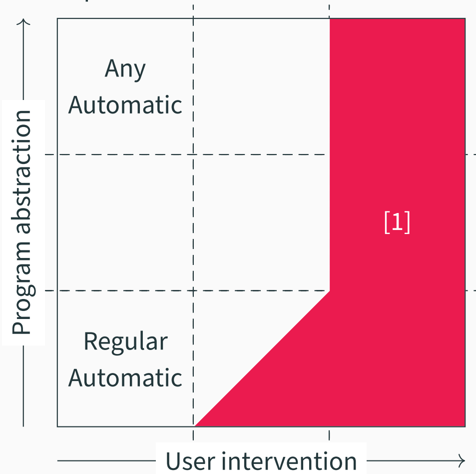
Current verification techniques vs. regular problems

Techniques classification:



Current verification techniques vs. regular problems

Techniques classification:



1. Proof assistants (Coq, Isabelle HOL, etc.):
    - Large range of properties
    - Requires a lot of manual work

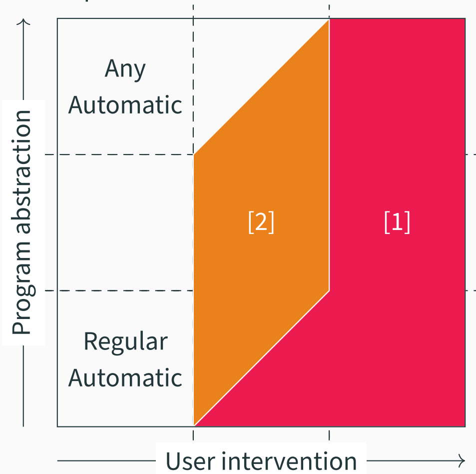
Inria. “The coq proof assistant”, 2016

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. “Isabelle/HOL – A Proof Assistant for Higher-Order Logic”, 2002.

...and more.

Current verification techniques vs. regular problems

Techniques classification:



1. Proof assistants (Coq, Isabelle HOL, etc.):
   - Large range of properties
   - Requires a lot of manual work
2. Refinement types (Liquid Types, F^*):
   - Only require some annotations
   - Reduced range of properties

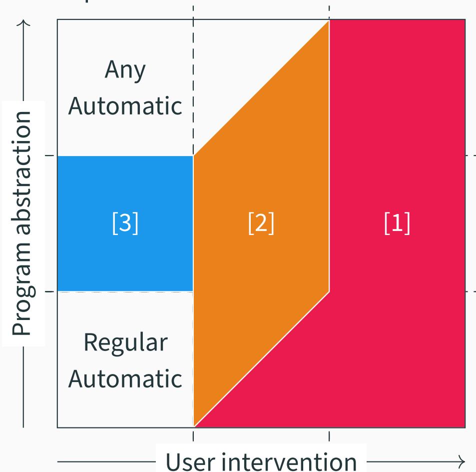
Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. "Liquid types", 2018

Microsoft Research and Inria. "F^* ", 2013.

...and more.

Current verification techniques vs. regular problems

Techniques classification:



3. HORS, Predicate abstraction, CHC:
    - No annotations needed
    - Focus on relational properties over numerical datatypes

C.-H. Luke Ong and Steven J. Ramsay. "Verifying higher-order functional programs with pattern-matching algebraic data types", 2006

Naoki Kobayashi. "Types and higher-order recursion schemes for verification of higher-order programs", 2009

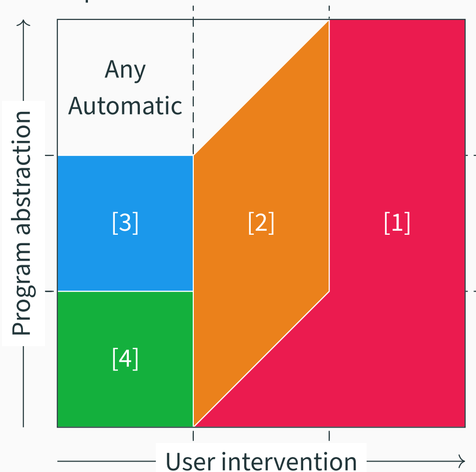
Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. "Predicate abstraction and CEGAR for higher-order model checking", 2010

Adrien Champion, Naoki Kobayashi, Ryosuke Sato. "HoIce: An ICE-Based Non-linear Horn Clause Solver", 2018

...and more.

Current verification techniques vs. regular problems

Techniques classification:



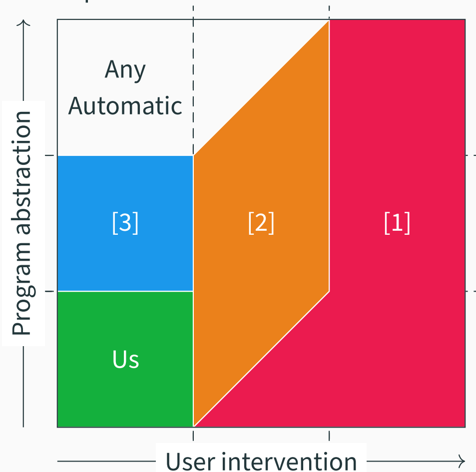
3. HORS, Predicate abstraction, CHC:
   - No annotations needed
   - Focus on relational properties over numerical datatypes
4. Regular language-based techniques:
   - Uncomplete (Timbuk)
   - Non-scalable (Matsumoto et al. 2015)

Yuma Matsumoto, Naoki Kobayashi, and Hiroshi Unno. “Automata-based abstraction for automated verification of higher-order tree-processing programs”, 2015

Thomas Genet, Timothée Haudebourg, and Thomas Jensen. “Verifying higher-order functions with tree automata”, 2018

Current verification techniques vs. regular problems

Techniques classification:



3. HORS, Predicate abstraction, CHC:
   - No annotations needed
   - Focus on relational properties over numerical datatypes
4. Regular language-based techniques:
   - Uncomplete (Timbuk)
   - Non-scalable (Matsumoto et al. 2015)

Yuma Matsumoto, Naoki Kobayashi, and Hiroshi Unno. “Automata-based abstraction for automated verification of higher-order tree-processing programs”, 2015

Thomas Genet, Timothée Haudebourg, and Thomas Jensen. “Verifying higher-order functions with tree automata”, 2018

Preliminaries

Terms, Rewriting Systems & Regular Languages

We use **terms** to represent the states and values of a program.

Definition (Term)

Let Σ be **ranked-alphabet**. A term defined over Σ is written:

$f \in \Sigma$ → sub-term →

$$f(t_1, \ldots, t_n)$$

← $ar(f) = n$

We write $ar(f)$ the **arity** of the symbol f.

Definition (Pattern)

Let $\mathcal{X}$ be a set of **variables**. We name **pattern** a term defined over $\Sigma \cup \mathcal{X}$.

Baader, F., Nipkow, T.. "Term Rewriting and All That", 1998

Hubert Comon et al. "Tree Automata Techniques and Applications", 2007

From OCaml to terms

Here are some examples of terms encoding OCaml expressions:

OCaml expression	Term
`true`	*true*
`false`	*false*
`0`	0
`1`	$s(0)$
`2`	$s(s(0))$
`[]`	*nil*
`a::[]`	$cons(a, nil)$
`a::b::[]`	$cons(a, cons(b, nil))$
`f a b`	$f(a, b)$
`if a then b else c`	$ite(a, b, c)$
`let g = function x -> x + 1 in body`	*body*

From OCaml to terms

Here are some examples of terms encoding OCaml expressions:

OCaml expression	Term
`true`	*true*
`false`	*false*
`0`	0
`1`	$s(0)$
`2`	$s(s(0))$
`[]`	*nil*
`a::[]`	$cons(a, nil)$
`a::b::[]`	$cons(a, cons(b, nil))$
`f a b`	$@(@(f, a), b)$
`if a then b else c`	$ite(a, b, c)$
`let g = function x -> x + 1 in body`	*body*

John Reynolds. "Automatic Computation of Data Set Definitions", 1969

Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. "Comparing Curried and Uncurried Rewriting", 1996

Term Rewriting Systems

We use **term rewriting systems** (TRSs) to represent the program itself.

Definition (Term Rewriting System)

A TRS $\mathcal{R}$ is a set of rewriting rules of the form

$$l \to r$$

where

- l and r are **patterns**
- $Var(r) \subseteq Var(l)$
- r is **linear** (for the purpose of the presentation)

TeReSe. "Term Rewriting Systems", 2011

Franz Baader, Tobias Nipkow. "Term Rewriting and All That", 1999

We write "$s \to_{\mathcal{R}} t$" when there exists a rule $l \to r$ in $\mathcal{R}$, a position p and a substitution σ such that $s|_p = l\sigma \wedge t = s[r\sigma]_p$.

We write $\to_{\mathcal{R}}^*$ the reflexive transitive closure of $\to_{\mathcal{R}}$.

From OCaml to TRSs

Here is an example of TRS encoding our previous `sorted` function:

```
let rec sorted cmp = function
    | _ -> true
    | x::y::l -> cmp x y && sorted (y::l)
```

This can be translated into the following TRS (variables are *underlined*):

From OCaml to TRSs

Here is an example of TRS encoding our previous `sorted` function:

```
let rec sorted cmp = function
    | _ -> true
    | x::y::l -> cmp x y && sorted (y::l)
```

This can be translated into the following TRS (variables are *underlined*):

$$
\begin{aligned}
@(@(sorted, \underline{cmp}), nil) &\rightarrow true \\
@(@(sorted, \underline{cmp}), cons(\underline{x}, nil)) &\rightarrow true \\
@(@(sorted, \underline{cmp}), cons(\underline{x}, cons(\underline{y}, \underline{l}))) &\rightarrow and(@(@(\underline{cmp}, \underline{x}), \underline{y}), @(@(sorted, \underline{cmp}), cons(\underline{y}, \underline{l})))
\end{aligned}
$$

From OCaml to TRSs

Here is an example of TRS encoding our previous `sorted` function:

```
let rec sorted cmp = function
    | _ -> true
    | x::y::l -> cmp x y && sorted (y::l)
```

This can be translated into the following TRS (variables are *underlined*):

$$\begin{aligned} \mathit{sorted}\ \underline{\mathit{cmp}}\ \mathit{nil} &\to \mathit{true} \\ \mathit{sorted}\ \underline{\mathit{cmp}}\ \mathit{cons}(\underline{x}, \mathit{nil}) &\to \mathit{true} \\ \mathit{sorted}\ \underline{\mathit{cmp}}\ \mathit{cons}(\underline{x}, \mathit{cons}(\underline{y}, \underline{l})) &\to \mathit{and}(\underline{\mathit{cmp}}\ \underline{x}\ \underline{y},\ \mathit{sorted}\ \underline{\mathit{cmp}}\ \mathit{cons}(\underline{y}, \underline{l})) \end{aligned}$$

We write “$f\ a\ b$” instead of “$@(@(f, a), b)$”...

Regular tree languages

A set of term is called a **tree language**.
It is **regular** iff it can be recognized by a **tree automaton**.

Definition (Bottom-Up Tree Automaton)

A tree automaton $\mathcal{A}$ is a quadruple $\langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ where

- Σ is a ranked alphabet
- $\mathcal{Q}$ a set of **states**
- $\mathcal{Q}_f \subseteq \mathcal{Q}$ a set of **final states**
- Δ a rewriting system of **transitions** of the form

Hubert Comon et al. "Tree Automata Techniques and Applications", 2007

$$f(q_1, \ldots, q_n) \to q \qquad \text{or} \qquad q' \to q \quad (\epsilon\text{-transition})$$

A term t is **recognized** by $\mathcal{A}$ if there exists a final state $q \in \mathcal{Q}_f$ such that $t \to_\Delta^* q$.
We write $\mathcal{L}(\mathcal{A})$ the language recognized by $\mathcal{A}$.

Regular tree languages

A set of term is called a **tree language**.

It is **regular** iff it can be recognized by a **tree automaton**.

Definition (Bottom-Up Tree Automaton)

A tree automaton $\mathcal{A}$ is a pair $\langle \mathcal{Q}, \Delta \rangle$ where

- Σ is the considered ranked alphabet
- $\mathcal{Q}$ a set of **states**
- $\mathcal{Q}_f = \mathcal{Q}$ a set of **final states**
- Δ a rewriting system of **transitions** of the form

$$f(q_1, \ldots, q_n) \rightarrow q \qquad \text{or} \qquad q' \rightarrow q \quad (\epsilon\text{-transition})$$

Hubert Comon et al. "Tree Automata Techniques and Applications", 2007

A term t is **recognized** by $\mathcal{A}$ if there exists a final state $q \in \mathcal{Q}_f$ such that $t \rightarrow^*_\Delta q$.

We write $\mathcal{L}(\mathcal{A})$ the language recognized by $\mathcal{A}$.

Regular Language Types

Regular languages can be used to represent OCaml types (and more).

For instance, the following OCaml type definition:

```
type ab = A | B ;;
```

...can be translated into the tree automaton

$$\langle \Sigma = \{ A : 0, B : 0 \}, \mathcal{Q} = \{ ab \}, \mathcal{Q}_f = \mathcal{Q}, \Delta \rangle$$

where Δ is defined by:

$$A \to ab \qquad B \to ab$$

Regular Language Types

Regular languages can be used to represent OCaml types (and more).

For instance, the following OCaml type definition:

```
type nat = 0 | S of nat ;;
```

...can be translated into the tree automaton

$$\langle \Sigma = \{ 0 : 0, S : 1 \}, \mathcal{Q} = \{ \mathit{nat} \}, \mathcal{Q}_f = \mathcal{Q}, \Delta \rangle$$

where Δ is defined by:

$$0 \rightarrow \mathit{nat} \qquad\qquad S(\mathit{nat}) \rightarrow \mathit{nat}$$

Regular Language Types

Regular languages can be used to represent OCaml types (and more).

For instance, the following OCaml type definition:

```
type nlist = Nil | Cons of nat * nlist ;;
```

...can be translated into the tree automaton

$$\langle \Sigma = \{ 0 : 0, S : 1, \mathit{Nil} : 0, \mathit{Cons} : 2 \}, \mathcal{Q} = \{ \mathit{nat}, \mathit{nlist} \}, \mathcal{Q}_f = \{ \mathit{nlist} \}, \Delta \rangle$$

where Δ is defined by:

$$0 \to \mathit{nat} \qquad \mathit{Nil} \to \mathit{nlist}$$
$$S(\mathit{nat}) \to \mathit{nat} \qquad \mathit{Cons}(\mathit{nat}, \mathit{nlist}) \to \mathit{nlist}$$

Regular Language Types

Regular languages can be used to represent OCaml types (and more).

For instance, the following OCaml type definition:

```
???
```

...can be translated into the tree automaton

$$\langle \Sigma = \{ 0 : 0, S : 1 \}, \mathcal{Q} = \{ \mathit{even}, \mathit{odd} \}, \mathcal{Q}_f = \{ \mathit{even} \}, \Delta \rangle$$

where Δ is defined by:

$$0 \rightarrow \mathit{even} \qquad S(\mathit{even}) \rightarrow \mathit{odd} \qquad S(\mathit{odd}) \rightarrow \mathit{even}$$

Regular Language Types

Regular languages can be used to represent OCaml types (and more).

For instance, the following OCaml type definition:

```
???
```

...can be translated into the tree automaton

$$\langle \Sigma = \{ 0 : 0, S : 1 \}, \mathcal{Q} = \{ \mathit{even}, \mathit{odd} \}, \mathcal{Q}_f = \{ \mathit{even} \}, \Delta \rangle$$

where Δ is defined by:

$$0 \to \mathit{even} \qquad S(\mathit{even}) \to \mathit{odd} \qquad S(\mathit{odd}) \to \mathit{even}$$

This is a **regular language type**

Regular Verification

Regular verification problem (OCaml perspective)

Input problem

```
insert : (ab -> ab -> bool) -> ab -> ab list -> ab list
sort : (ab -> ab -> bool) -> ab list -> ab list
sorted : (ab -> ab -> bool) -> ab list -> bool
```

```
sorted cmp (sort cmp (l : ab list)) : { true }
```

Output

```
insert : { cmp } -> ab -> { sorted lists } -> { sorted lists }
sort : { cmp } -> ab list -> { sorted list }
sorted : { cmp } -> { sorted lists } -> { true }
```

...with the definitions of `{ cmp }`, `{ true }` and `{ sorted lists }`.

Regular verification problem (example)

Input problem

TRS $\mathcal{R}$ (the program):

$$insert\ \underline{cmp}\ \underline{x}\ \underline{l} \to \ldots$$

$$sort\ \underline{cmp}\ \underline{l} \to \ldots$$

$$sorted\ \underline{cmp}\ \underline{l} \to \ldots$$

Property: $\forall \underline{l}.\ sorted\ cmp\ (sort\ cmp\ \underline{l}) \to^*_{\mathcal{R}} true$

Output

An TRS abstraction $\mathcal{R}^\#$ of $\mathcal{R}$:

$$insert\ cmp^\#\ ab^\#\ sorted_lists^\# \to sorted_lists^\#$$

$$sort\ cmp^\#\ ab_list^\# \to sorted_lists^\#$$

$$sorted\ cmp^\#\ sorted_lists^\# \to true^\#$$

...with the definition of $cmp^\#$, $true^\#$, $sorted_lists^\#$ with an automaton $\Delta^\#$ such that:

$$sorted\ cmp^\#\ (sort\ cmp^\#\ ab_list^\#) \to^*_{\mathcal{R}^\# \cup \Delta} true^\#$$

Regular verification problem (general)

Input problem

- A TRS $\mathcal{R}$ defined over Σ
- A pattern p with π mapping each variable of p to a language
- A target output language O

$$\forall \sigma, \sigma(x) \in \pi(x). \quad p\sigma \to_{\mathcal{R}}^{*} v \Rightarrow v \in O \qquad \text{with } v \in IRR(\mathcal{R})$$

Output

- An abstraction $\mathcal{R}^{\#}$ of $\mathcal{R}$
- An abstract domain (tree automaton) $\Lambda = \langle \Sigma^{\#}, \Delta^{\#} \rangle$ such that
  - $O^{\#} \in \Sigma^{\#}$ with $O = \mathcal{L}(\Lambda, O^{\#})$
  - for all $x \in Var(p)$, $\sigma(x)^{\#} \in \Sigma^{\#}$ with $\sigma(x) = \mathcal{L}(\Lambda, O^{\#})$
  - $p\sigma^{\#} \to_{\mathcal{R}^{\#} \cup \Delta^{\#}}^{*} O^{\#}$ where $\sigma^{\#}(x) = \sigma(x)^{\#}$.

Abstraction

Definition (TRS Abstraction)

An abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ in $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$:

- rules of the form "$f(a_1^\#, \ldots, a_n^\#) \to a^\#$" with $a_i^\#, a^\# \in \Sigma^\#$
- defines a simulation:

$$\begin{array}{ccc} s & \longrightarrow & t \\ \wr & & \wr \\ s^\# & \dashrightarrow & t^\# \end{array}$$

$\mathcal{R}^\#$ can be seen as a set of **type signatures** for the program's functions.

Abstraction

Definition (TRS Abstraction)

An abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ in $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$:

- rules of the form "$f(a_1^\#, \ldots, a_n^\#) \to a^\#$" with $a_i^\#, a^\# \in \Sigma^\#$
- defines a simulation:

$$\begin{array}{ccc} s & \xrightarrow{\mathcal{R}} & t \\ {\scriptstyle \mathcal{R}^\# \cup \Delta^\#} \downarrow {\scriptstyle *} & & {\scriptstyle *} \downarrow {\scriptstyle \mathcal{R}^\# \cup \Delta^\#} \\ s^\# & = & t^\# \end{array}$$

Note: $\mathcal{R}^\# \cup \Delta^\#$ is not necessarily deterministic (multiple abstractions for one term).

$\mathcal{R}^\#$ can be seen as a set of **type signatures** for the program's functions.

Abstraction inference using a backward typing algorithm

Algorithm

1. **Extract a template signature for the topmost symbol**
   - The output type is always known
   - Constant inputs are known
2. **Analyse function**
   - Using this algorithm recursively or;
   - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**

$$\forall \underline{l}.\ sorted\ cmp\ (sort\ cmp\ \underline{l}) \rightarrow^*_{\mathcal{R}} true$$

Abstraction inference using a backward typing algorithm

Algorithm

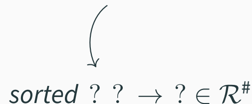
1. **Extract a template signature for the topmost symbol**
    - The output type is always known
    - Constant inputs are known
2. **Analyse function**
    - Using this algorithm recursively or;
    - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**

$$sorted\ cmp\ (sort\ (\underline{l}\ :\ ablist^{\#}))\ :\ true^{\#}$$

Abstraction inference using a backward typing algorithm

Algorithm

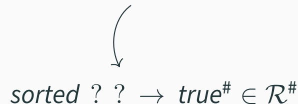
1. **Extract a template signature for the topmost symbol**
   - The output type is always known
   - Constant inputs are known
2. **Analyse function**
   - Using this algorithm recursively or;
   - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**



Abstraction inference using a backward typing algorithm

Algorithm

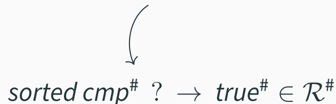
1. **Extract a template signature for the topmost symbol**
   - The output type is always known
   - Constant inputs are known
2. **Analyse function**
   - Using this algorithm recursively or;
   - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**



Abstraction inference using a backward typing algorithm

Algorithm

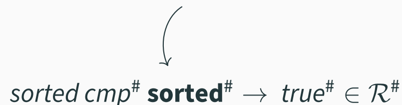
1. **Extract a template signature for the topmost symbol**
    - The output type is always known
    - Constant inputs are known
2. **Analyse function**
    - Using this algorithm recursively or;
    - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**



Abstraction inference using a backward typing algorithm

Algorithm

1. **Extract a template signature for the topmost symbol**
    - The output type is always known
    - Constant inputs are known
2. **Analyse function**
    - Using this algorithm recursively or;
    - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**



Abstraction inference using a backward typing algorithm

Algorithm

1. **Extract a template signature for the topmost symbol**
   - The output type is always known
   - Constant inputs are known
2. **Analyse function**
   - Using this algorithm recursively or;
   - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**

$$\mathit{sorted}\ \mathit{cmp}\ (\mathit{sort}\ (\underline{l}\ :\ \mathit{ablist}^{\#})\ :\ \mathit{sorted}^{\#})\ :\ \mathit{true}^{\#}$$

$$\mathit{sorted}\ \mathit{cmp}^{\#}\ \mathbf{sorted}^{\#} \rightarrow \mathit{true}^{\#} \in \mathcal{R}^{\#}$$

Abstraction inference using a backward typing algorithm

Algorithm

1. **Extract a template signature for the topmost symbol**
   - The output type is always known
   - Constant inputs are known
2. **Analyse function**
   - Using this algorithm recursively or;
   - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**

$$\mathit{sorted}\ \mathit{cmp}\ (\mathit{sort}\ (\underline{l}\ :\ \mathit{ablist}^\#)\ :\ \mathit{sorted}^\#)\ :\ \mathit{true}^\#$$

$$\mathit{sorted}\ \mathit{cmp}^\#\ \textbf{sorted}^\# \rightarrow \mathit{true}^\# \in \mathcal{R}^\#$$

$$\mathit{sort}\ (\underline{l}\ :\ \mathit{ablist}^\#)\ :\ \mathit{sorted}^\#$$

Abstraction inference using a backward typing algorithm

Algorithm

1. **Extract a template signature for the topmost symbol**
   - The output type is always known
   - Constant inputs are known
2. **Analyse function**
   - Using this algorithm recursively or;
   - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**

$$\mathit{sorted}\ \mathit{cmp}\ (\mathit{sort}\ (\underline{l}\ :\ \mathit{ablist}^{\#})\ :\ \mathit{sorted}^{\#})\ :\ \mathit{true}^{\#}$$

$$\mathit{sorted}\ \mathit{cmp}^{\#}\ \textbf{sorted}^{\#} \rightarrow \mathit{true}^{\#} \in \mathcal{R}^{\#}$$

$$\mathit{sort}\ (\underline{l}\ :\ \mathit{ablist}^{\#})\ :\ \mathit{sorted}^{\#}$$

$$\mathit{sort}\ ?\ \rightarrow\ ? \in \mathcal{R}^{\#}$$

Abstraction inference using a backward typing algorithm

Algorithm

1. **Extract a template signature for the topmost symbol**
   - The output type is always known
   - Constant inputs are known
2. **Analyse function**
   - Using this algorithm recursively or;
   - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**

$$\mathit{sorted}\ \mathit{cmp}\ (\mathit{sort}\ (\underline{l} : \mathit{ablist}^{\#}) : \mathit{sorted}^{\#}) : \mathit{true}^{\#}$$

$$\mathit{sorted}\ \mathit{cmp}^{\#}\ \mathbf{sorted}^{\#} \rightarrow \mathit{true}^{\#} \in \mathcal{R}^{\#}$$

$$\mathit{sort}\ (\underline{l} : \mathit{ablist}^{\#}) : \mathit{sorted}^{\#}$$

$$\mathit{sort}\ \mathit{ablist}^{\#} \rightarrow \mathit{sorted}^{\#} \in \mathcal{R}^{\#}$$

Abstraction inference using a backward typing algorithm

Algorithm

1. **Extract a template signature for the topmost symbol**
   - The output type is always known
   - Constant inputs are known
2. **Analyse function**
   - Using this algorithm recursively or;
   - Fixpoint learning procedure.
3. **Check**
4. **Repeat with the sub-terms**

$$\mathit{sorted}\ \mathit{cmp}\ (\mathit{sort}\ (\underline{l}\ :\ \mathit{ablist}^{\#})\ :\ \mathit{sorted}^{\#})\ :\ \mathit{true}^{\#}$$

$$\mathit{sorted}\ \mathit{cmp}^{\#}\ \mathbf{sorted}^{\#} \to \mathit{true}^{\#} \in \mathcal{R}^{\#}$$

$$\mathit{sort}\ (\underline{l}\ :\ \mathit{ablist}^{\#})\ :\ \mathit{sorted}^{\#}$$

$$\mathit{sort}\ \mathit{ablist}^{\#} \to \mathit{sorted}^{\#} \in \mathcal{R}^{\#}$$

Function analysis

Input

- The TRS $\mathcal{R}$
- The function's symbol $f \in \Sigma$
- The target output language O

Output

- A set of rules of the form:

$$f(I_1^\#, \ldots, I_n^\#) \to O^\#$$

...a **complete** abstraction of the function.

Function analysis

Input

- The TRS $\mathcal{R}$
- The function's symbol $f \in \Sigma$
- The target output language O

Example

- $ite(true, \underline{x}, \underline{y}) \to \underline{x}$
  $ite(false, \underline{x}, \underline{y}) \to \underline{y}$
- ite
- $\{A\}$

Output

- A set of rules of the form:

$$f(I_1^\#, \ldots, I_n^\#) \to O^\#$$

...a **complete** abstraction of the function.

Example

Non-recursive case: computed using the **previous backward typing algorithm**.

Function analysis

Input	**Example**
• The TRS $\mathcal{R}$ • The function's symbol $f \in \Sigma$ • The target output language O	• $ite(true, \underline{x}, \underline{y}) \rightarrow \underline{x}$ $ite(false, \underline{x}, \underline{y}) \rightarrow \underline{y}$ • ite • $\{A\}$

Output	**Example**
• A set of rules of the form: $$f(I_1^\#, \ldots, I_n^\#) \rightarrow O^\#$$ ...a **complete** abstraction of the function.	$$ite(true, \underline{x}, \underline{y}) \rightarrow \underline{x}$$ Non-recursive case: computed using the **previous backward typing algorithm**.

Function analysis

Input	Example
• The TRS $\mathcal{R}$ • The function's symbol $f \in \Sigma$ • The target output language O	• $ite(true, \underline{x}, \underline{y}) \to \underline{x}$ $ite(false, \underline{x}, \underline{y}) \to \underline{y}$ • ite • $\{A\}$

Output	Example
• A set of rules of the form: $f(I_1^\#, \ldots, I_n^\#) \to O^\#$ ...a **complete** abstraction of the function.	$ite(true, \underline{x}, \underline{y}) \to \underline{x} : A^\#$ Non-recursive case: computed using the **previous backward typing algorithm**.

Function analysis

Input

- The TRS $\mathcal{R}$
- The function's symbol $f \in \Sigma$
- The target output language O

Example

- $ite(true, \underline{x}, \underline{y}) \to \underline{x}$
  $ite(false, \underline{x}, \underline{y}) \to \underline{y}$
- ite
- $\{A\}$

Output

- A set of rules of the form:

$$f(I_1^\#, \ldots, I_n^\#) \to O^\#$$

...a **complete** abstraction of the function.

Example

$$ite(true, \underline{x} : A^\#, \underline{y}) \to \underline{x} : A^\#$$

Non-recursive case: computed using the **previous backward typing algorithm**.

Function analysis

Input

- The TRS $\mathcal{R}$
- The function's symbol $f \in \Sigma$
- The target output language O

Example

- $ite(true, \underline{x}, \underline{y}) \to \underline{x}$
  $ite(false, \underline{x}, \underline{y}) \to \underline{y}$
- ite
- $\{A\}$

Output

- A set of rules of the form:

$$f(I_1^\#, \dots, I_n^\#) \to O^\#$$

...a **complete** abstraction of the function.

Example

$$ite(true, \underline{x} : A^\#, \underline{y} : \top^\#) \to \underline{x} : A^\#$$

Non-recursive case: computed using the **previous backward typing algorithm**.

Function analysis

Input	**Example**
• The TRS $\mathcal{R}$ • The function's symbol $f \in \Sigma$ • The target output language O	• $ite(true, \underline{x}, \underline{y}) \to \underline{x}$ $ite(false, \underline{x}, \underline{y}) \to \underline{y}$ • ite • $\{A\}$

Output	**Example**
• A set of rules of the form: $$f(I_1^\#, \ldots, I_n^\#) \to O^\#$$ ...a **complete** abstraction of the function.	$$ite(true, A^\#, \top^\#) \to A^\#$$ Non-recursive case: computed using the **previous backward typing algorithm**.

Function analysis

Input

- The TRS $\mathcal{R}$
- The function's symbol $f \in \Sigma$
- The target output language O

Example

- $ite(true, \underline{x}, \underline{y}) \to \underline{x}$
  $ite(false, \underline{x}, \underline{y}) \to \underline{y}$
- ite
- $\{A\}$

Output

- A set of rules of the form:

$$f(I_1^\#, \ldots, I_n^\#) \to O^\#$$

...a **complete** abstraction of the function.

Example

$$ite(true, A^\#, \top^\#) \to A^\#$$
$$ite(false^\#, \top^\#, A^\#) \to A^\#$$

Non-recursive case: computed using the **previous backward typing algorithm**.

Function analysis

Input

- The TRS $\mathcal{R}$
- The function's symbol $f \in \Sigma$
- The target output language O

Example

- $\mathit{even}(0) \to \mathit{true} \qquad \mathit{even}(s(0)) \to \mathit{false}$
  $$\mathit{even}(s(s(\underline{x}))) \to \mathit{even}(\underline{x})$$
- even
- $\{\mathit{true}\}$

Output

- A set of rules of the form:

$$f(I_1^\#, \dots, I_n^\#) \to O^\#$$

...a **complete** abstraction of the function.

Example

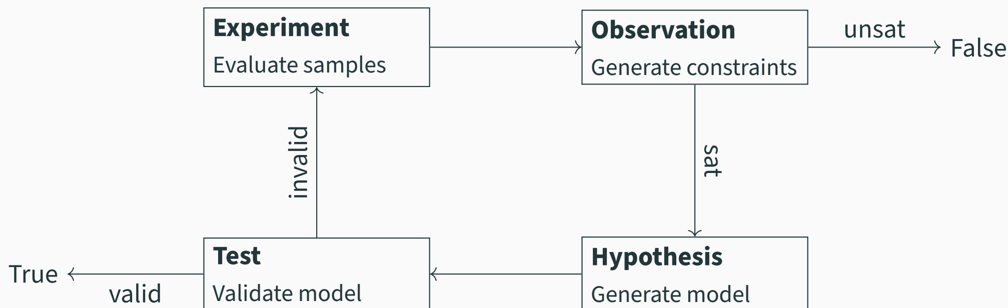
$$\mathit{even}(\mathit{Even}^\#) \to \mathit{true}^\#$$

Recursive-case: we need a **regular invariant learning procedure**.

Regular Invariant Learning

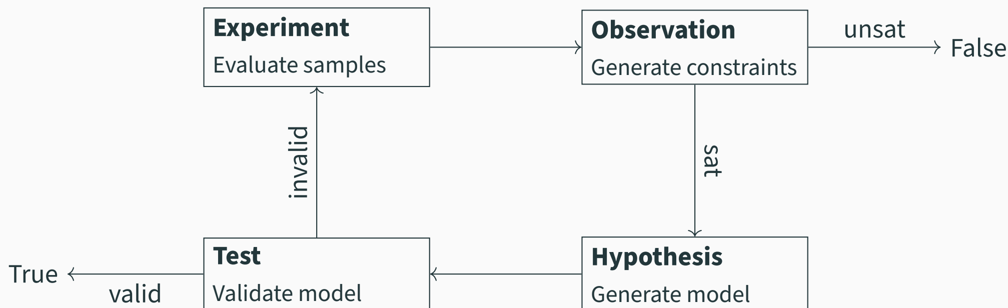
Regular Invariant Learning Procedure

A 4 steps SMT-based CEGAR-like procedure: the scientific method!



Regular Invariant Learning Procedure

A 4 steps SMT-based CEGAR-like procedure: the scientific method!



Note: In our setting a counter example is a term $f(t_1, \ldots, t_n)$ that rewrites into $O^{\#}$ **and** $\overline{O}^{\#}$.

Step 1: Experiment

Definition (Experimental automaton $\mathcal{A}_i$)

- Finite, deterministic, reduced
- Recognizes a fragment of $\{ f(t_1, \ldots, t_n) \}$
- $\mathcal{R}$-closed using the **Tree Automaton Completion algorithm**

$\mathcal{A}_i$ contains all our **knowledge** about the function f at iteration i.

Step 1: Experiment

Definition (Experimental automaton $\mathcal{A}_i$)

- Finite, deterministic, reduced
- Recognizes a fragment of $\{ f(t_1, \ldots, t_n) \}$
- $\mathcal{R}$-closed using the **Tree Automaton Completion algorithm**

$\mathcal{A}_i$ contains all our **knowledge** about the function f at iteration i.

Example

Consider $\mathcal{A}_0$ recognizing the terms $even(0)$ and $even(s(s(0)))$:

$$0 \to q_0 \qquad s(q_0) \to q_1 \qquad s(q_1) \to q_2$$
$$even(q_0) \to q_{f0} \qquad even(q_1) \to q_{f1} \qquad even(q_2) \to q_{f2}$$

Step 1: Experiment

Definition (Experimental automaton $\mathcal{A}_i$)

- Finite, deterministic, reduced
- Recognizes a fragment of $\{ f(t_1, \dots, t_n) \}$
- $\mathcal{R}$-closed using the **Tree Automaton Completion algorithm**

$\mathcal{A}_i$ contains all our **knowledge** about the function f at iteration i.

Example

Consider $\mathcal{A}_0$ recognizing the terms $even(0)$ and $even(s(s(0)))$:

$$
\begin{array}{rclrclrcl}
true & \to & q_{true} & false & \to & q_{false} & & & \\
0 & \to & q_0 & s(q_0) & \to & q_1 & s(q_1) & \to & q_2 \\
even(q_0) & \to & q_{f0} & even(q_1) & \to & q_{f1} & even(q_2) & \to & q_{f2} \\
q_{true} & \to & q_{f0} & q_{false} & \to & q_{f1} & q_{true} & \to & q_{f2}
\end{array}
$$

Step 2: Observation (1/2)

Definition (SMT-constraints)

For all abstract value $v \in \mathcal{L}(O^\#)$ and $v' \in \mathcal{L}(\overline{O}^\#)$ such that

$$\begin{array}{ccc} f(t_1, \ldots, t_n) \xrightarrow[\mathcal{R}]{*} v & \qquad & f(t'_1, \ldots, t'_n) \xrightarrow[\mathcal{R}]{*} v' \\ \mathcal{A}_i \downarrow * & & \mathcal{A}_i \downarrow * \\ q & & q' \end{array}$$

Step 2: Observation (1/2)

Definition (SMT-constraints)

For all abstract value $v \in \mathcal{L}(O^{\#})$ and $v' \in \mathcal{L}(\overline{O}^{\#})$ such that

$$
\begin{array}{ccc}
f(t_1, \ldots, t_n) \xrightarrow[\mathcal{R}]{*} v & & f(t'_1, \ldots, t'_n) \xrightarrow[\mathcal{R}]{*} v' \\
\mathcal{A}_i \downarrow * & & \mathcal{A}_i \downarrow * \\
q & \neq & q'
\end{array}
$$

We generate the SMT-constraint $q \neq q'$.

Step 2: Observation (1/2)

Definition (SMT-constraints)

For all abstract value $v \in \mathcal{L}(O^{\#})$ and $v' \in \mathcal{L}(\overline{O}^{\#})$ such that

$$\begin{array}{ccc} f(t_1, \ldots, t_n) \xrightarrow[\mathcal{R}]{*} v & & f(t'_1, \ldots, t'_n) \xrightarrow[\mathcal{R}]{*} v' \\ \mathcal{A}_i \downarrow * & & \mathcal{A}_i \downarrow * \\ q & \neq & q' \end{array}$$

We generate the SMT-constraint $q \neq q'$.

Example

$$\begin{array}{ccc} even(0) \xrightarrow[\mathcal{R}]{*} true & & even(s(0)) \xrightarrow[\mathcal{R}]{*} false \\ \mathcal{A}_i \downarrow * & & \mathcal{A}_i \downarrow * \\ q_{f0} & \neq & q_{f1} \end{array}$$

Step 2: Observation (2/2)

Definition (Determinism constraints)

For all abstract value $v \in \mathcal{L}(O^{\#})$ and $v' \in \mathcal{L}(\overline{O}^{\#})$ such that

$$f(q_1, \ldots, q_n) \to q \qquad f(q'_1, \ldots, q'_n) \to q'$$

We generate the SMT-constraint

$$q \neq q' \Rightarrow q_1 \neq q'_1 \vee \cdots \vee q_2 \neq q'_2$$

Step 2: Observation (2/2)

Definition (Determinism constraints)

For all abstract value $v \in \mathcal{L}(O^\#)$ and $v' \in \mathcal{L}(\overline{O}^\#)$ such that

$$f(q_1, \ldots, q_n) \to q \qquad\qquad f(q'_1, \ldots, q'_n) \to q'$$

We generate the SMT-constraint

$$q \neq q' \Rightarrow q_1 \neq q'_1 \vee \cdots \vee q_2 \neq q'_2$$

Example

$$even(q_0) \to q_{f0} \qquad\qquad even(q_1) \to q_{f1}$$

We generate the SMT-constraint

$$q_{f0} \neq q_{f1} \Rightarrow q_0 \neq q_1$$

Step 2: Observation (2/2)

Definition (Determinism constraints)

For all abstract value $v \in \mathcal{L}(O^\#)$ and $v' \in \mathcal{L}(\overline{O}^\#)$ such that

$$f(q_1, \ldots, q_n) \to q \qquad\qquad f(q'_1, \ldots, q'_n) \to q'$$

We generate the SMT-constraint

$$q \neq q' \Rightarrow q_1 \neq q'_1 \vee \cdots \vee q_2 \neq q'_2$$

Example

$$even(q_0) \to q_{f0} \qquad\qquad even(q_1) \to q_{f1}$$

We generate the SMT-constraint

$$q_0 \neq q_1$$

Step 3: Hypothesis

Definition (Candidate abstraction generation)

We generate $\mathcal{A}_i^\# = \phi(\mathcal{A}_i)$ where ϕ is a solution to our SMT-constraints.

$$\phi : \mathcal{Q} \mapsto \Sigma^\#$$

ϕ maps each concrete state to an abstract value.

Step 3: Hypothesis

Definition (Candidate abstraction generation)

We generate $\mathcal{A}_i^{\#} = \phi(\mathcal{A}_i)$ where ϕ is a solution to our SMT-constraints.

$$\phi : \mathcal{Q} \mapsto \Sigma^{\#}$$

ϕ maps each concrete state to an abstract value.

Example

$$
\begin{aligned}
q_{true} &\neq q_{false} \\
q_{f0} &\neq q_{f1} \\
q_{f1} &\neq q_{f2} \\
q_{0} &\neq q_{1} \\
q_{1} &\neq q_{2}
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
q_{true} &\mapsto a^{\#} \\
q_{f0} &\mapsto a^{\#} \\
q_{f2} &\mapsto a^{\#} \\
q_{0} &\mapsto a^{\#} \\
q_{2} &\mapsto a^{\#}
\end{aligned}
\qquad
\begin{aligned}
q_{false} &\mapsto b^{\#} \\
q_{f1} &\mapsto b^{\#} \\
q_{1} &\mapsto b^{\#}
\end{aligned}
$$

Step 3: Hypothesis

Definition (Candidate abstraction generation)

We generate $\mathcal{A}_i^\# = \phi(\mathcal{A}_i)$ where ϕ is a solution to our SMT-constraints.

$$\phi : \mathcal{Q} \mapsto \Sigma^\#$$

ϕ maps each concrete state to an abstract value.

Example

$$\begin{aligned} q_{true} &\neq q_{false} \\ q_{f0} &\neq q_{f1} \\ q_{f1} &\neq q_{f2} \\ q_0 &\neq q_1 \\ q_1 &\neq q_2 \end{aligned} \quad \Rightarrow \quad \begin{aligned} q_{true} &\mapsto true^\# & q_{false} &\mapsto false^\# \\ q_{f0} &\mapsto true^\# & q_{f1} &\mapsto false^\# \\ q_{f2} &\mapsto true^\# & q_1 &\mapsto Odd^\# \\ q_0 &\mapsto Even^\# \\ q_2 &\mapsto Even^\# \end{aligned}$$

Step 3: Hypothesis

Definition (Candidate abstraction generation)

We generate $\mathcal{A}_i^\# = \phi(\mathcal{A}_i)$ where ϕ is a solution to our SMT-constraints.

$$\phi : \mathcal{Q} \mapsto \Sigma^\#$$

ϕ maps each concrete state to an abstract value.

Example

$$\begin{array}{rcl} \mathit{true} \rightarrow q_{\mathit{true}} & \mathit{false} \rightarrow q_{\mathit{false}} & \\ 0 \rightarrow q_0 & s(q_0) \rightarrow q_1 & s(q_1) \rightarrow q_2 \\ \mathit{even}(q_0) \rightarrow q_{f0} & \mathit{even}(q_1) \rightarrow q_{f1} & \mathit{even}(q_2) \rightarrow q_{f2} \\ q_{\mathit{true}} \rightarrow q_{f0} & q_{\mathit{false}} \rightarrow q_{f1} & q_{\mathit{true}} \rightarrow q_{f2} \end{array}$$

Step 3: Hypothesis

Definition (Candidate abstraction generation)

We generate $\mathcal{A}_i^{\#} = \phi(\mathcal{A}_i)$ where ϕ is a solution to our SMT-constraints.

$$\phi : \mathcal{Q} \mapsto \Sigma^{\#}$$

ϕ maps each concrete state to an abstract value.

Example

$$\begin{aligned} true &\to true^{\#} & false &\to false^{\#} & & \\ 0 &\to Even^{\#} & s(Even^{\#}) &\to Odd^{\#} & s(Odd^{\#}) &\to Even^{\#} \\ even(Even^{\#}) &\to true^{\#} & even(Odd^{\#}) &\to false^{\#} & even(Even^{\#}) &\to Odd^{\#} \\ true^{\#} &\to true^{\#} & false^{\#} &\to false^{\#} & true^{\#} &\to true^{\#} \end{aligned}$$

Step 3: Hypothesis

Definition (Candidate abstraction generation)

We generate $\mathcal{A}_i^\# = \phi(\mathcal{A}_i)$ where ϕ is a solution to our SMT-constraints.

$$\phi : \mathcal{Q} \mapsto \Sigma^\#$$

ϕ maps each concrete state to an abstract value.

Example

$$\begin{array}{rrr}
true \rightarrow true^\# & false \rightarrow false^\# & \\
0 \rightarrow Even^\# & s(Even^\#) \rightarrow Odd^\# & s(Odd^\#) \rightarrow Even^\# \\
even(Even^\#) \rightarrow true^\# & even(Odd^\#) \rightarrow false^\# & \\
true^\# \rightarrow true^\# & false^\# \rightarrow false^\# &
\end{array}$$

Step 3: Hypothesis

Definition (Candidate abstraction generation)

We generate $\mathcal{A}_i^\# = \phi(\mathcal{A}_i)$ where ϕ is a solution to our SMT-constraints.

$$\phi : \mathcal{Q} \mapsto \Sigma^\#$$

ϕ maps each concrete state to an abstract value.

Example

$$\begin{array}{rrr} \mathit{true} \rightarrow \mathit{true}^\# & \mathit{false} \rightarrow \mathit{false}^\# & \\ 0 \rightarrow \mathit{Even}^\# & s(\mathit{Even}^\#) \rightarrow \mathit{Odd}^\# & s(\mathit{Odd}^\#) \rightarrow \mathit{Even}^\# \\ \mathit{even}(\mathit{Even}^\#) \rightarrow \mathit{true}^\# & \mathit{even}(\mathit{Odd}^\#) \rightarrow \mathit{false}^\# & \end{array}$$

Step 4: Test

Termination

$\mathcal{A}_i^{\#}$ contains a valid abstraction $\mathcal{R}^{\#}$ and $\Lambda = \langle \Sigma^{\#}, \Delta^{\#} \rangle$ when:

- it is $\mathcal{R}$**-closed**
- it is **complete**

Otherwise we go back to Step 1: Experiment (with new samples).

Step 4: Test

Termination

$\mathcal{A}_i^\#$ contains a valid abstraction $\mathcal{R}^\#$ and $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ when:

- it is $\mathcal{R}$-**closed**
- it is **complete**

Otherwise we go back to Step 1: Experiment (with new samples).

Example

Our previous candidate model is a valid abstraction:

$$\begin{array}{rlrlrll} \mathit{true} &\to \mathit{true}^\# & \mathit{false} &\to \mathit{false}^\# & & & \\ 0 &\to \mathit{Even}^\# & s(\mathit{Even}^\#) &\to \mathit{Odd}^\# & s(\mathit{Odd}^\#) &\to \mathit{Even}^\# & \text{this is } \Delta^\# \\ \mathit{even}(\mathit{Even}^\#) &\to \mathit{true}^\# & \mathit{even}(\mathit{Odd}^\#) &\to \mathit{false}^\# & & & \text{this is } \mathcal{R}^\# \end{array}$$

Step 4: Test

$\mathcal{R}$

$$even(0) \rightarrow true \qquad even(s(0)) \rightarrow false$$

$$even(s(s(\underline{x}))) \rightarrow \underline{x}$$

Example

Our previous candidate model is a valid abstraction:

$$true \rightarrow true^{\#} \qquad false \rightarrow false^{\#}$$

$$0 \rightarrow Even^{\#} \qquad s(Even^{\#}) \rightarrow Odd^{\#} \qquad s(Odd^{\#}) \rightarrow Even^{\#} \qquad \text{this is } \Delta^{\#}$$

$$even(Even^{\#}) \rightarrow true^{\#} \qquad even(Odd^{\#}) \rightarrow false^{\#} \qquad \text{this is } \mathcal{R}^{\#}$$

Properties of the Invariant Learning Procedure

Regular Completeness

If there exists a **regular** abstraction $\mathcal{R}^\#, \Delta^\#$ that separates $O^\#$ and $\overline{O}^\#$, we will eventually find it.

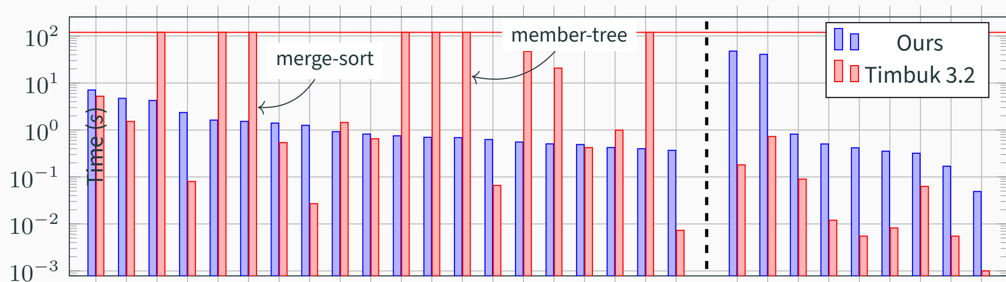
Completeness in Refutation

If there exists a counter example (a term t that rewrites into both $O^\#$ and $\overline{O}^\#$), then we will eventually find it.

Experiments

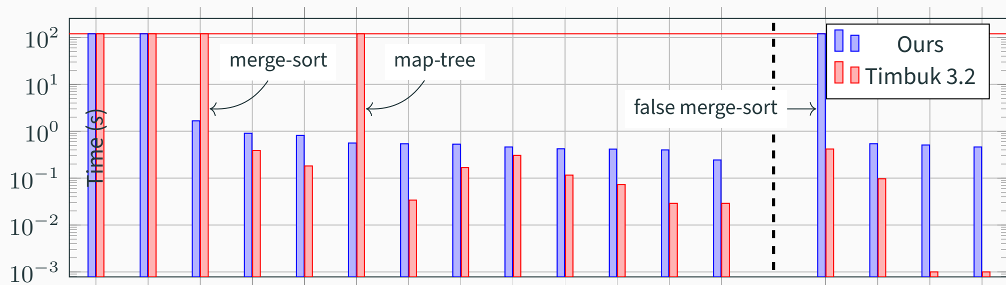
Experiments

Implementation in Timbuk 4, tested on 80+ problems, compared with Timbuk 3.



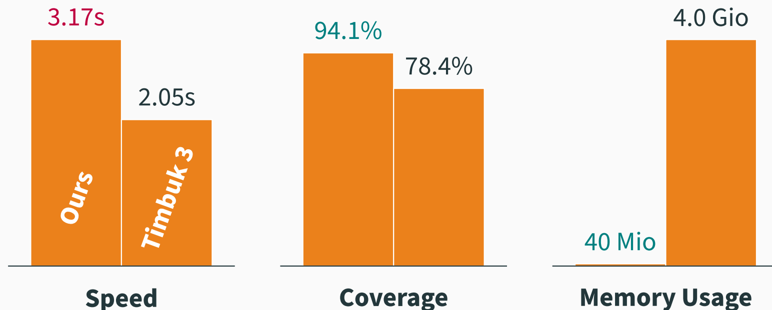
Experiments

Implementation in Timbuk 4, tested on 80+ problems, compared with Timbuk 3.



Conclusion

Implementation in Timbuk 4, tested on 80+ problems, compared with Timbuk 3.



- “Regularly” complete, complete in refutation.
- Modular thanks to the type system approach.

https://gitlab.inria.fr/regular-pv/timbuk/timbuk