Static Analysis of Functional Programs using Tree Automata

Thomas Genet & Yann Salmon

INRIA/IRISA/Université de Rennes 1

Outline

1. Motivating example
2. Background on tree automata completion
3. What is missing for a decent static analysis of functional programs?

... Related work scattered in subsections

Motivating example

OCaml type checking

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | h::t -> h :: (append t l2);;
# val append: 'a list -> 'a list -> 'a list = <fun>

let rec rev l = match l with
  | [] -> []
  | h::t -> append (rev t) [h];;
# val rev: 'a list -> 'a list = <fun>
```

Motivating example

OCaml type checking

```
let rec append l1 l2 = match l1 with
  | [] -> []
  | h::t -> h :: (append t l2);;
# val append: 'a list -> 'a list -> 'a list = <fun>

let rec rev l = match l with
  | [] -> []
  | h::t -> append (rev t) [h];;
# val rev: 'a list -> 'a list = <fun>
```

Motivating example

OCaml type checking

```
let rec append l1 l2 = match l1 with
  | [] -> []
  | h::t -> h :: (append t l2);;
# val append: 'a list -> 'a list -> 'a list = <fun>

let rec rev l = match l with
  | [] -> []
  | h::t -> append (rev t) [h];;
# val rev: 'a list -> 'a list = <fun>
```

We would like to have... more than simple types

```
# val rev: 'a list -> empty list
```

Motivating example (II)

OCaml type checking

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | h::t -> h :: (append t l2);;

let rec rev l = match l with
  | [] -> []
  | h::t -> append (rev t) [h];;
```

Motivating example (II)

OCaml type checking

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | h::t -> h :: (append t l2);;

let rec rev l = match l with
  | [] -> []
  | h::t -> append (rev t) [h];;
```

We would like to have...

```
val rev: list of As then Bs -> list of Bs then As
```

Background: Term Rewriting

Sets of symbols and variables

- Set of ranked symbols $\mathcal{F} = \{app : 2, cons : 2, nil : 0, a : 0\}$
- Set of variables $\mathcal{X} = \{x, y, z, \ldots\}$

Background: Term Rewriting

Sets of symbols and variables

- Set of ranked symbols $\mathcal{F} = \{app : 2, cons : 2, nil : 0, a : 0\}$
- Set of variables $\mathcal{X} = \{x, y, z, \ldots\}$

Sets of terms

- Ground terms $\mathcal{T}(\mathcal{F}) = \{a,\ nil,\ cons(a, nil),\ cons(a, cons(a, nil)),\ \ldots\}$
- Terms $\mathcal{T}(\mathcal{F}, \mathcal{X}) = \{x,\ cons(x, y),\ app(nil, a),\ \ldots\}$

Background: Term Rewriting

Sets of symbols and variables

- Set of ranked symbols $\mathcal{F} = \{app : 2, cons : 2, nil : 0, a : 0\}$
- Set of variables $\mathcal{X} = \{x, y, z, \ldots\}$

Sets of terms

- Ground terms $\mathcal{T}(\mathcal{F}) = \{a,\ nil,\ cons(a, nil),\ cons(a, cons(a, nil)),\ \ldots\}$
- Terms $\mathcal{T}(\mathcal{F}, \mathcal{X}) = \{x,\ cons(x, y),\ app(nil, a),\ \ldots\}$

Term Rewriting Systems (TRS)

Set of rewrite rules $l \to r$ with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $Var(r) \subseteq Var(l)$ *e.g.*

$$\mathcal{R} = \left\{ \begin{array}{l} app(nil, x) \to x \\ app(cons(x, y), z) \to cons(x, app(y, z)) \end{array} \right\}$$

Background: Term Rewriting

Sets of symbols and variables

- Set of ranked symbols $\mathcal{F} = \{app : 2, cons : 2, nil : 0, a : 0\}$
- Set of variables $\mathcal{X} = \{x, y, z, \ldots\}$

Sets of terms

- Ground terms $\mathcal{T}(\mathcal{F}) = \{a,\ nil,\ cons(a, nil),\ cons(a, cons(a, nil)),\ \ldots\}$
- Terms $\mathcal{T}(\mathcal{F}, \mathcal{X}) = \{x,\ cons(x, y),\ app(nil, a),\ \ldots\}$

Term Rewriting Systems (TRS)

Set of rewrite rules $l \rightarrow r$ with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $Var(r) \subseteq Var(l)$ *e.g.*

$$\mathcal{R} = \left\{ \begin{array}{l} app(nil, x) \rightarrow x \\ app(cons(x, y), z) \rightarrow cons(x, app(y, z)) \end{array} \right\}$$

Term Rewriting (II)

- Rewriting term $app(cons(a, nil), cons(b, nil))$ using

$$\mathcal{R} = \left\{ \begin{array}{l} app(nil, x) \rightarrow x \\ app(cons(x, y), z) \rightarrow cons(x, app(y, z)) \end{array} \right\}$$

Term Rewriting (II)

- Rewriting term $app(cons(a, nil), cons(b, nil))$ using

$$\mathcal{R} = \left\{ \begin{array}{l} app(nil, x) \to x \\ app(cons(x, y), z) \to cons(x, app(y, z)) \end{array} \right\}$$

$$app(cons(a, nil), cons(b, nil)) \quad \to_{\mathcal{R}} cons(a, app(nil, cons(b, nil)))$$

Term Rewriting (II)

- Rewriting term $app(cons(a, nil), cons(b, nil))$ using

$$\mathcal{R} = \left\{ \begin{array}{l} app(nil, x) \to x \\ app(cons(x, y), z) \to cons(x, app(y, z)) \end{array} \right\}$$

$$\begin{array}{rl} app(cons(a, nil), cons(b, nil)) & \to_{\mathcal{R}} cons(a, app(nil, cons(b, nil))) \\ & \to_{\mathcal{R}} cons(a, cons(b, nil)) \end{array}$$

Term Rewriting (II)

- Rewriting term $app(cons(a, nil), cons(b, nil))$ using

$$\mathcal{R} = \left\{ \begin{array}{l} app(nil, x) \to x \\ app(cons(x, y), z) \to cons(x, app(y, z)) \end{array} \right\}$$

$$\begin{array}{rl} app(cons(a, nil), cons(b, nil)) & \to_{\mathcal{R}} cons(a, app(nil, cons(b, nil))) \\ & \to_{\mathcal{R}} cons(a, cons(b, nil)) \end{array}$$

- Set of reachable terms: $\mathcal{R}^*(\mathcal{L}) = \{u \mid s \in \mathcal{L} \wedge s \to_{\mathcal{R}}^* u\}$

Term Rewriting (II)

- Rewriting term $app(cons(a, nil), cons(b, nil))$ using

$$\mathcal{R} = \left\{ \begin{array}{l} app(nil, x) \to x \\ app(cons(x, y), z) \to cons(x, app(y, z)) \end{array} \right\}$$

$$\begin{array}{rl} app(cons(a, nil), cons(b, nil)) & \to_{\mathcal{R}} cons(a, app(nil, cons(b, nil))) \\ & \to_{\mathcal{R}} cons(a, cons(b, nil)) \end{array}$$

- Set of reachable terms: $\mathcal{R}^*(\mathcal{L}) = \{u \mid s \in \mathcal{L} \wedge s \to_{\mathcal{R}}{}^* u\}$

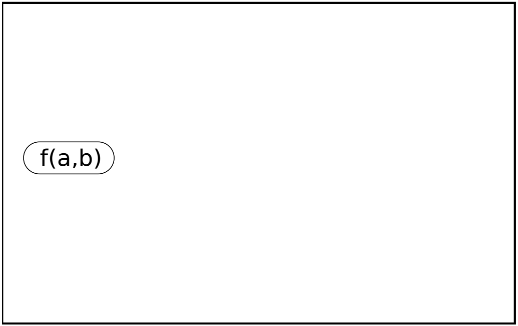
$\mathcal{R}^*(\{app(cons(a, nil), cons(b, nil))\}) =$

$$\{ \begin{array}{l} app(cons(a, nil), cons(b, nil)), \\ cons(a, app(nil, cons(b, nil))), \\ cons(a, cons(b, nil)) \end{array} \}$$

Equational abstraction [Meseguer & al. 03] [Takai 04]

$$\mathcal{R} = \begin{cases} (1)\ f(x,y) \rightarrow f(g(x),y) \\ (2)\ f(x,y) \rightarrow f(x,h(y)) \end{cases}$$

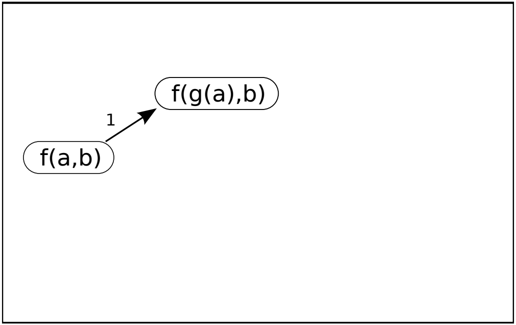
prove that $f(a,b) \not\rightarrow_{\mathcal{R}}{}^{*} f(a,h(g(b)))$?



Equational abstraction [Meseguer & al. 03] [Takai 04]

$$\mathcal{R} = \begin{cases} (1)\ f(x,y) \to f(g(x),y) \\ (2)\ f(x,y) \to f(x,h(y)) \end{cases}$$

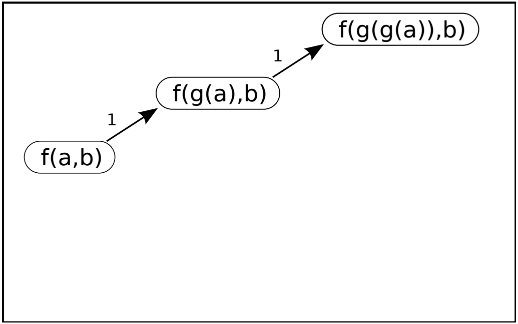
prove that $f(a,b) \not\to_{\mathcal{R}}^{*} f(a,h(g(b)))$?



Equational abstraction [Meseguer & al. 03] [Takai 04]

$$\mathcal{R} = \begin{cases} (1)\ f(x,y) \to f(g(x),y) \\ (2)\ f(x,y) \to f(x,h(y)) \end{cases}$$

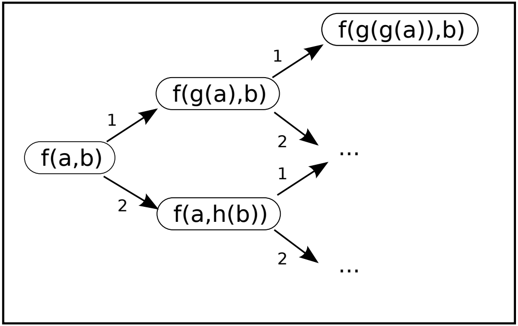
prove that $f(a,b) \not\to_{\mathcal{R}}^{*} f(a,h(g(b)))$?



Equational abstraction [Meseguer & al. 03] [Takai 04]

$$\mathcal{R} = \begin{cases} (1)\ f(x,y) \to f(g(x),y) \\ (2)\ f(x,y) \to f(x,h(y)) \end{cases}$$

prove that $f(a,b) \not\to_{\mathcal{R}}^{*} f(a,h(g(b)))$?

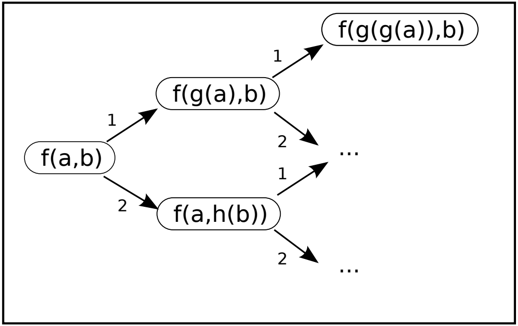


Equational abstraction [Meseguer & al. 03] [Takai 04]

$$\mathcal{R} = \begin{cases} (1)\ f(x,y) \rightarrow f(g(x),y) \\ (2)\ f(x,y) \rightarrow f(x,h(y)) \end{cases}$$

prove that $f(a,b) \not\rightarrow_{\mathcal{R}}^{*} f(a,h(g(b)))$?

using $E = \{g(g(x)) = g(x), h(h(x)) = h(x)\}$

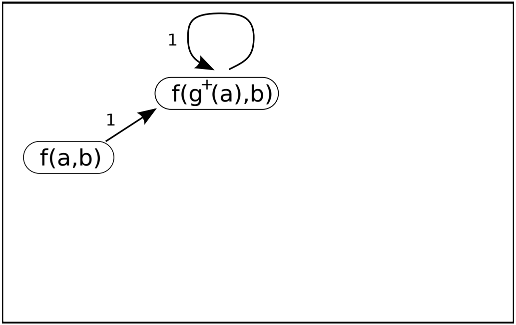


Equational abstraction [Meseguer & al. 03] [Takai 04]

$$\mathcal{R} = \begin{cases} (1)\ f(x,y) \rightarrow f(g(x),y) \\ (2)\ f(x,y) \rightarrow f(x,h(y)) \end{cases}$$

prove that $f(a,b) \not\rightarrow_{\mathcal{R}}^{*} f(a,h(g(b)))$?

using $E = \{g(g(x)) = g(x), h(h(x)) = h(x)\}$

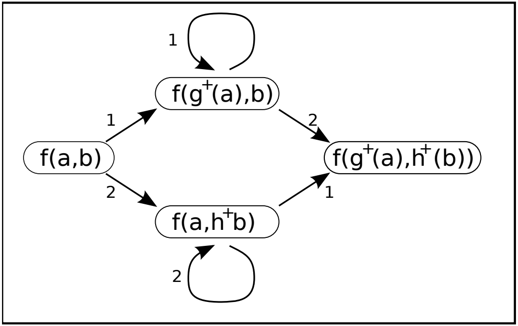


Equational abstraction [Meseguer & al. 03] [Takai 04]

$$\mathcal{R} = \begin{cases} (1)\ f(x,y) \to f(g(x),y) \\ (2)\ f(x,y) \to f(x,h(y)) \end{cases}$$

prove that $f(a,b) \not\to_{\mathcal{R}}^{*} f(a,h(g(b)))$?

using $E = \{g(g(x)) = g(x), h(h(x)) = h(x)\}$

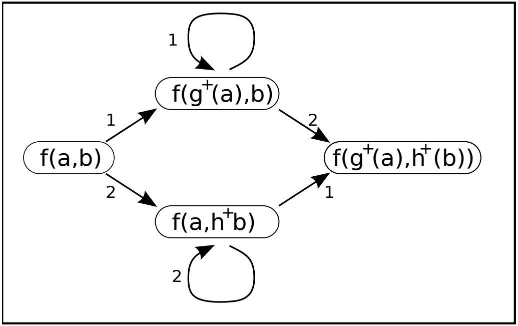


Equational abstraction [Meseguer & al. 03] [Takai 04]

$$\mathcal{R} = \begin{cases} (1)\ f(x,y) \to f(g(x),y) \\ (2)\ f(x,y) \to f(x,h(y)) \end{cases}$$

prove that $f(a,b) \not\to_{\mathcal{R}}^{*} f(a,h(g(b)))$?

using $E = \{g(g(x)) = g(x), h(h(x)) = h(x)\}$



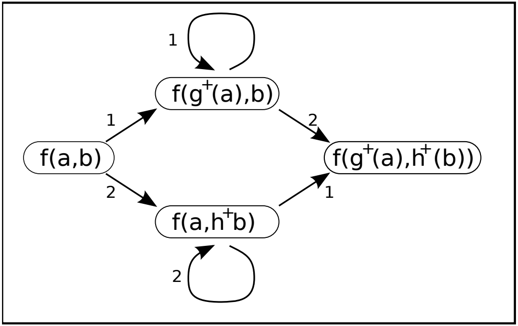
$$f(a,b) \not\to_{\mathcal{R}/E}^{*} f(a,h(g(b)))$$

Equational abstraction [Meseguer & al. 03] [Takai 04]

$$\mathcal{R} = \begin{cases} (1)\ f(x,y) \to f(g(x),y) \\ (2)\ f(x,y) \to f(x,h(y)) \end{cases}$$

prove that $f(a,b) \not\to_{\mathcal{R}}^{*} f(a,h(g(b)))$?

using $E = \{g(g(x)) = g(x), h(h(x)) = h(x)\}$



$$f(a,b) \not\to_{\mathcal{R}/E}^{*} f(a,h(g(b))) \quad \Rightarrow \quad f(a,b) \not\to_{\mathcal{R}}^{*} f(a,h(g(b)))$$

Background: Tree Automata

Recognized language $\mathcal{L}(\mathcal{A}, q)$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

$$\Delta = \begin{cases} a \to q_a \\ b \to q_b \\ \\ nil \to q_{la} \\ nil \to q_{lb} \\ \\ cons(q_a, q_{la}) \to q_{la} \\ cons(q_b, q_{lb}) \to q_{lb} \\ \\ app(q_{la}, q_{lb}) \to q_f \end{cases}$$

Background: Tree Automata

Recognized language $\mathcal{L}(\mathcal{A}, q) = \{s \in \mathcal{T}(\mathcal{F}) \mid s \rightarrow^*_\Delta q\}$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

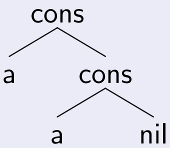
$$\Delta = \begin{cases} a \rightarrow q_a \\ b \rightarrow q_b \\ \\ nil \rightarrow q_{la} \\ nil \rightarrow q_{lb} \\ \\ cons(q_a, q_{la}) \rightarrow q_{la} \\ cons(q_b, q_{lb}) \rightarrow q_{lb} \\ \\ app(q_{la}, q_{lb}) \rightarrow q_f \end{cases}$$

Background: Tree Automata

Recognized language $\mathcal{L}(\mathcal{A}, q) = \{s \in \mathcal{T}(\mathcal{F}) \mid s \rightarrow^*_{\Delta} q\}$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

$$\Delta = \begin{cases} a \rightarrow q_a \\ b \rightarrow q_b \\ \\ nil \rightarrow q_{la} \\ nil \rightarrow q_{lb} \\ \\ cons(q_a, q_{la}) \rightarrow q_{la} \\ cons(q_b, q_{lb}) \rightarrow q_{lb} \\ \\ app(q_{la}, q_{lb}) \rightarrow q_f \end{cases}$$

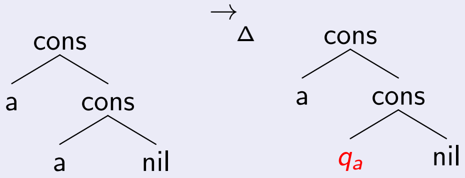


Background: Tree Automata

Recognized language $\mathcal{L}(\mathcal{A}, q) = \{s \in \mathcal{T}(\mathcal{F}) \mid s \to_{\Delta}^{*} q\}$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

$$\Delta = \begin{cases} a \to q_a \\ b \to q_b \\ \\ nil \to q_{la} \\ nil \to q_{lb} \\ \\ cons(q_a, q_{la}) \to q_{la} \\ cons(q_b, q_{lb}) \to q_{lb} \\ \\ app(q_{la}, q_{lb}) \to q_f \end{cases}$$

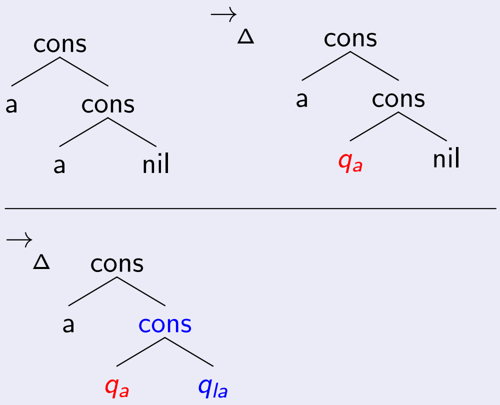


Background: Tree Automata

Recognized language $\mathcal{L}(\mathcal{A}, q) = \{s \in \mathcal{T}(\mathcal{F}) \mid s \rightarrow^*_{\Delta} q\}$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

$$\Delta = \begin{cases} a \rightarrow q_a \\ b \rightarrow q_b \\ \\ nil \rightarrow q_{la} \\ nil \rightarrow q_{lb} \\ \\ cons(q_a, q_{la}) \rightarrow q_{la} \\ cons(q_b, q_{lb}) \rightarrow q_{lb} \\ \\ app(q_{la}, q_{lb}) \rightarrow q_f \end{cases}$$

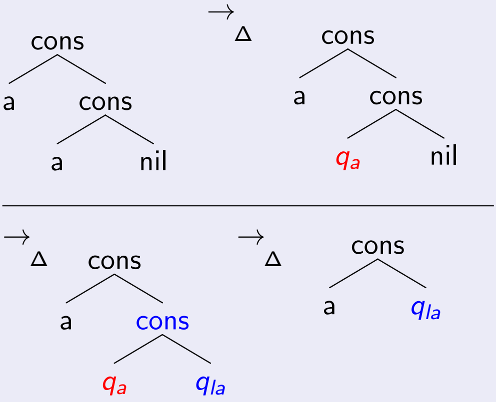


Background: Tree Automata

Recognized language $\mathcal{L}(\mathcal{A}, q) = \{s \in \mathcal{T}(\mathcal{F}) \mid s \rightarrow_{\Delta}^{*} q\}$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

$$\Delta = \begin{cases} a \rightarrow q_a \\ b \rightarrow q_b \\ \\ nil \rightarrow q_{la} \\ nil \rightarrow q_{lb} \\ \\ cons(q_a, q_{la}) \rightarrow q_{la} \\ cons(q_b, q_{lb}) \rightarrow q_{lb} \\ \\ app(q_{la}, q_{lb}) \rightarrow q_f \end{cases}$$

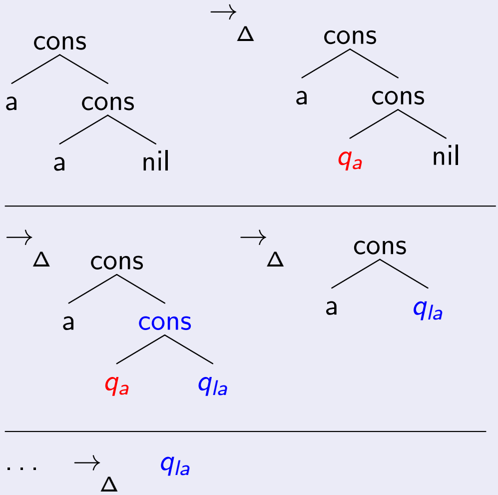


Background: Tree Automata

Recognized language $\mathcal{L}(\mathcal{A}, q) = \{s \in \mathcal{T}(\mathcal{F}) \mid s \rightarrow^*_{\Delta} q\}$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

$$\Delta = \begin{cases} a \rightarrow q_a \\ b \rightarrow q_b \\ \\ nil \rightarrow q_{la} \\ nil \rightarrow q_{lb} \\ \\ cons(q_a, q_{la}) \rightarrow q_{la} \\ cons(q_b, q_{lb}) \rightarrow q_{lb} \\ \\ app(q_{la}, q_{lb}) \rightarrow q_f \end{cases}$$



Tree Automata (II)

Recognized language: $\mathcal{L}(\mathcal{A}, q) = \{s \mid s \rightarrow^*_{\Delta} q\}$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

$$\Delta = \begin{cases} a \rightarrow q_a \\ b \rightarrow q_b \\ \\ nil \rightarrow q_{la} \\ nil \rightarrow q_{lb} \\ \\ cons(q_a, q_{la}) \rightarrow q_{la} \\ cons(q_b, q_{lb}) \rightarrow q_{lb} \\ \\ app(q_{la}, q_{lb}) \rightarrow q_f \end{cases}$$

$\mathcal{L}(\mathcal{A}, q_{la}) = \{nil, cons(a, nil), cons(a, \ldots)\}$

Tree Automata (II)

Recognized language: $\mathcal{L}(\mathcal{A}, q) = \{s \mid s \rightarrow^*_{\Delta} q\}$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

$$\Delta = \begin{cases} a \rightarrow q_a \\ b \rightarrow q_b \\ \\ nil \rightarrow q_{la} \\ nil \rightarrow q_{lb} \\ \\ cons(q_a, q_{la}) \rightarrow q_{la} \\ cons(q_b, q_{lb}) \rightarrow q_{lb} \\ \\ app(q_{la}, q_{lb}) \rightarrow q_f \end{cases}$$

$\mathcal{L}(\mathcal{A}, q_{la}) = \{nil, cons(a, nil), cons(a, ...)\}$

$\mathcal{L}(\mathcal{A}, q_{lb}) = \{nil, cons(b, nil), cons(b, ...)\}$

Tree Automata (II)

Recognized language: $\mathcal{L}(\mathcal{A}, q) = \{s \mid s \xrightarrow[\Delta]{*} q\}$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

$$\Delta = \begin{cases} a \to q_a \\ b \to q_b \\ \\ nil \to q_{la} \\ nil \to q_{lb} \\ \\ cons(q_a, q_{la}) \to q_{la} \\ cons(q_b, q_{lb}) \to q_{lb} \\ \\ app(q_{la}, q_{lb}) \to q_f \end{cases}$$

$$\mathcal{L}(\mathcal{A}, q_{la}) = \{nil, cons(a, nil), cons(a, \ldots)\}$$

$$\mathcal{L}(\mathcal{A}, q_{lb}) = \{nil, cons(b, nil), cons(b, \ldots)\}$$

$$\mathcal{L}(\mathcal{A}, q_f) = \{app(la, lb) \mid la \in \mathcal{L}(\mathcal{A}, q_{la}) \wedge lb \in \mathcal{L}(\mathcal{A}, q_{lb})\}$$

Tree Automata (II)

Recognized language: $\mathcal{L}(\mathcal{A}, q) = \{s \mid s \to_\Delta^* q\}$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
with $\mathcal{Q} = \{q_a, q_b, q_{la}, q_{lb}, q_f\}$
$\mathcal{Q}_f = \{q_f\}$ and

$$\Delta = \begin{cases} a \to q_a \\ b \to q_b \\ \\ nil \to q_{la} \\ nil \to q_{lb} \\ \\ cons(q_a, q_{la}) \to q_{la} \\ cons(q_b, q_{lb}) \to q_{lb} \\ \\ app(q_{la}, q_{lb}) \to q_f \end{cases}$$

$$\mathcal{L}(\mathcal{A}, q_{la}) = \{nil, cons(a, nil), cons(a, \ldots)\}$$

$$\mathcal{L}(\mathcal{A}, q_{lb}) = \{nil, cons(b, nil), cons(b, \ldots)\}$$

$$\mathcal{L}(\mathcal{A}, q_f) = \{app(la, lb) \mid la \in \mathcal{L}(\mathcal{A}, q_{la}) \wedge lb \in \mathcal{L}(\mathcal{A}, q_{lb})\}$$

$$\mathcal{L}(\mathcal{A}) = \{s \in \mathcal{T}(\mathcal{F}) \mid s \to_\Delta^* q \wedge q \in \mathcal{Q}_f\}$$

Tree Automata Completion to approximate $\mathcal{R}^*(\mathcal{L})$

Tree automata completion semi-algorithm (particular ARTMC)

- Input: a TRS $\mathcal{R}$, a tree automaton $\mathcal{A}$ and approximation equations E
- Output: an automaton $\mathcal{A}^*_{\mathcal{R},E}$

Tree Automata Completion to approximate $\mathcal{R}^*(\mathcal{L})$

Tree automata completion semi-algorithm (particular ARTMC)

- Input: a TRS $\mathcal{R}$, a tree automaton $\mathcal{A}$ and approximation equations E
- Output: an automaton $\mathcal{A}^*_{\mathcal{R},E}$

$$\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \subseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$$ [with V. Rusu, 2010]

Theorem 1 (Upper bound)

Given a left-linear TRS $\mathcal{R}$, a tree automaton $\mathcal{A}$ and a set of equations E, if completion terminates on $\mathcal{A}^*_{\mathcal{R},E}$ then $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.

Theorem 2 (Lower bound)

Given a left-linear TRS $\mathcal{R}$, a tree automaton $\mathcal{A}$ and a set of equations E, if $\mathcal{A}$ is R/E-coherent then $\mathcal{L}(\mathcal{A}^i_{\mathcal{R},E}) \subseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$.

Tree Automata Completion Demo : Timbuk

[with V. Viet Triem Tong, Y. Boichut, B. Boyer, V. Murat]
(Around 13000 lines of Ocaml)

Timbuk provides

- Tree automata completion
- Equational approximations
- Coq checker for completion results
- Beta: CEGAR, Abstract Domains (*e.g. integer intervals*)

Used for Cyptographic Protocol, Java and JavaScript verification

Demo:

- `demo_basic.txt`
- `demo_reverseBug.txt`

What is missing for static analysis of functional languages?

1. Define equations guaranteeing termination of completion
2. Deal with higher order functions
3. Take evaluation strategies into account
   - call by value (*e.g.* Ocaml) $\approx$ innermost rewrite strategy
   - call by need (*e.g.* Haskell) $\approx$ outermost rewrite strategy + sharing
   - order in pattern matching $\approx$ priority rewrite strategy
4. Deal with built-in types (`int`,`float`,`char`,`strings`, ...)
5. Have a modular analysis
6. Have a user friendly way to display/define language annotations ...

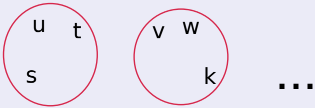
What is missing for static analysis of functional languages?

1. Define equations guaranteeing termination of completion
2. Deal with higher order functions
3. Take evaluation strategies into account
   - call by value (*e.g.* Ocaml) $\approx$ innermost rewrite strategy
   - call by need (*e.g.* Haskell) $\approx$ outermost rewrite strategy + sharing
   - order in pattern matching $\approx$ priority rewrite strategy
4. Deal with built-in types (`int`,`float`,`char`,`strings`, ...)
5. Have a modular analysis
6. Have a user friendly way to display/define language annotations ...

Equations guaranteeing termination of completion

Intuition: finite set of E-equivalence classes $\Rightarrow$ completion terminates

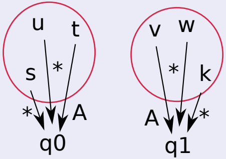
$\mathcal{T}(\mathcal{F})/_{=_E}$ is



Equations guaranteeing termination of completion

Intuition: finite set of E-equivalence classes $\Rightarrow$ completion terminates

$\mathcal{T}(\mathcal{F})/_{=_E}$ is

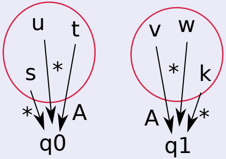


...

Equations guaranteeing termination of completion

Intuition: finite set of E-equivalence classes $\Rightarrow$ completion terminates

$\mathcal{T}(\mathcal{F})/_{=_E}$ is



...

Finite set of states

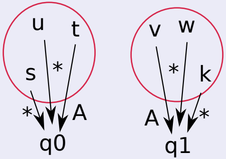
$\Rightarrow$

terminating completion

Equations guaranteeing termination of completion

Intuition: finite set of E-equivalence classes $\Rightarrow$ completion terminates

$\mathcal{T}(\mathcal{F})/_{=_E}$ is



…

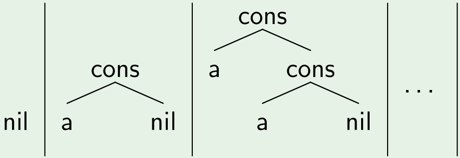
Finite set of states

$\Rightarrow$

terminating completion

Example 3 (Equations for the append function)

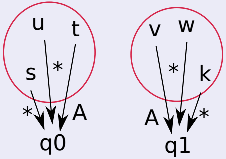
Let $\mathcal{F} = \{app : 2, cons : 2, nil : 0, a : 0\}$.



Equations guaranteeing termination of completion

Intuition: finite set of E-equivalence classes $\Rightarrow$ completion terminates

$\mathcal{T}(\mathcal{F})/_{=_E}$ is



...

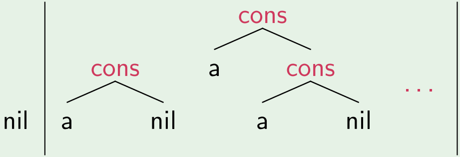
Finite set of states

$\Rightarrow$

terminating completion

Example 3 (Equations for the append function)

Let $\mathcal{F} = \{app : 2, cons : 2, nil : 0, a : 0\}$.

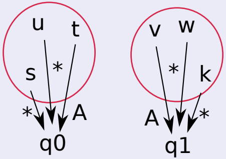


With $E =$
$\{cons(x, cons(y, z)) = cons(x, z)\}$

Equations guaranteeing termination of completion

Intuition: finite set of E-equivalence classes $\Rightarrow$ completion terminates

$\mathcal{T}(\mathcal{F})/_{=_E}$ is



...

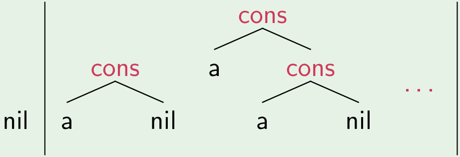
Finite set of states

$\Rightarrow$

terminating completion

Example 3 (Equations for the append function)

Let $\mathcal{F} = \{app : 2, cons : 2, nil : 0, a : 0\}$.



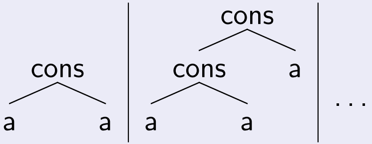
With $E = \{cons(x, cons(y, z)) = cons(x, z)\}$

But $\mathcal{T}(\mathcal{F})/_{=_E}$ is not finite!

Equations guaranteeing termination of completion (II)

With $E = \{cons(x, cons(y, z)) = cons(x, z)\}$, $\mathcal{T}(\mathcal{F})/_{=_E}$ is not finite!

Infinitely many classes of ill-typed terms



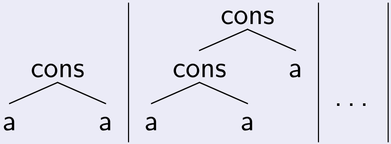
are all in different classes!

Ill-typed terms incompatible with `cons:`α `->` α `list ->` α `list`

Equations guaranteeing termination of completion (II)

With $E = \{cons(x, cons(y, z)) = cons(x, z)\}$, $\mathcal{T}(\mathcal{F})/_{=_E}$ is not finite!

Infinitely many classes of ill-typed terms



are all in different classes!

Ill-typed terms incompatible with

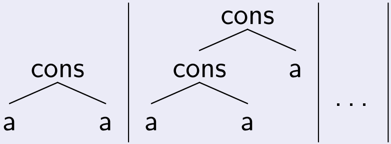
`cons:`α `->` α `list ->` α `list`

We restrict to well-typed terms $\mathcal{T}(\mathcal{F})^{\mathcal{S}}$

Equations guaranteeing termination of completion (II)

With $E = \{cons(x, cons(y,z)) = cons(x,z)\}$, $\mathcal{T}(\mathcal{F})/_{=_E}$ is not finite!

Infinitely many classes of ill-typed terms



are all in different classes!

Ill-typed terms incompatible with `cons:`α `->` α `list ->` α `list`

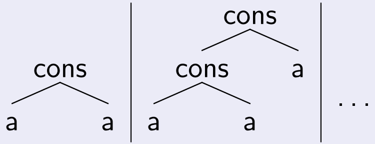
We restrict to well-typed terms $\mathcal{T}(\mathcal{F})^{\mathcal{S}}$

With $E = \{cons(x, cons(y,z)) = cons(x,z)\}$, $\mathcal{T}(\mathcal{F})^{\mathcal{S}}/_{=_E}$ is not finite!

Equations guaranteeing termination of completion (II)

With $E = \{cons(x, cons(y, z)) = cons(x, z)\}$, $\mathcal{T}(\mathcal{F})/_{=_E}$ is not finite!

Infinitely many classes of ill-typed terms



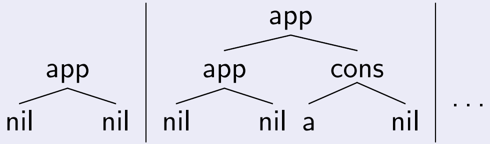
are all in different classes!

Ill-typed terms incompatible with `cons:`α `->` α `list ->` α `list`

We restrict to well-typed terms $\mathcal{T}(\mathcal{F})^{\mathcal{S}}$

With $E = \{cons(x, cons(y, z)) = cons(x, z)\}$, $\mathcal{T}(\mathcal{F})^{\mathcal{S}}/_{=_E}$ is not finite!

Infinitely many classes of partially evaluated terms



are all in different classes!

Partially evaluated terms

Equations guaranteeing termination of completion (III)

Proposed solution: use $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$

- $\mathcal{D}$efined and $\mathcal{C}$onstructor *e.g.* $\mathcal{D} = \{app\}$ and $\mathcal{C} = \{a, cons, nil\}$

Equations guaranteeing termination of completion (III)

Proposed solution: use $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$

- $\mathcal{D}$efined and $\mathcal{C}$onstructor *e.g.* $\mathcal{D} = \{app\}$ and $\mathcal{C} = \{a, cons, nil\}$
- Define $E^c_{\mathcal{C}}$ a set of contracting equations on $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$, such that $\mathcal{T}(\mathcal{C})^{\mathcal{S}}/_{=E^c_{\mathcal{C}}}$ is finite, *e.g.* $cons(x, cons(y, z)) = cons(x, z)$

Equations guaranteeing termination of completion (III)

Proposed solution: use $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$

- $\mathcal{D}$efined and $\mathcal{C}$onstructor *e.g.* $\mathcal{D} = \{app\}$ and $\mathcal{C} = \{a, cons, nil\}$
- Define $E^c_{\mathcal{C}}$ a set of contracting equations on $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$, such that $\mathcal{T}(\mathcal{C})^{\mathcal{S}}/_{=E^c_{\mathcal{C}}}$ is finite, *e.g.* $cons(x, cons(y, z)) = cons(x, z)$
- Define $E_{\mathcal{R}} = \{l = r \mid l \to r \in \mathcal{R}\}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$

Equations guaranteeing termination of completion (III)

Proposed solution: use $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$

- $\mathcal{D}$efined and $\mathcal{C}$onstructor *e.g.* $\mathcal{D} = \{app\}$ and $\mathcal{C} = \{a, cons, nil\}$
- Define $E^c_{\mathcal{C}}$ a set of contracting equations on $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$, such that $\mathcal{T}(\mathcal{C})^{\mathcal{S}}/_{=E^c_{\mathcal{C}}}$ is finite, *e.g.* $cons(x, cons(y, z)) = cons(x, z)$
- Define $E_{\mathcal{R}} = \{l = r \mid l \rightarrow r \in \mathcal{R}\}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$
- Theorem: If $\mathcal{R}$ is sufficiently complete then $\mathcal{T}(\mathcal{F})^{\mathcal{S}}/_{=E}$ is finite

$\mathcal{R}$ Sufficiently complete:

$\forall s \in \mathcal{T}(\mathcal{F})^{\mathcal{S}}. \exists t \in \mathcal{T}(\mathcal{C})^{\mathcal{S}}. s \rightarrow_{\mathcal{R}}{}^{*}\ t$

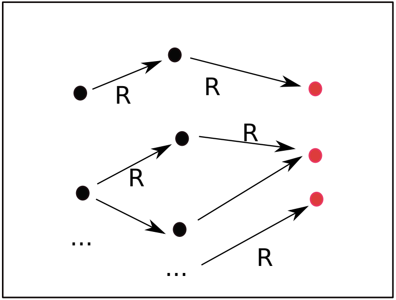
Equations guaranteeing termination of completion (III)

Proposed solution: use $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$

- $\mathcal{D}$efined and $\mathcal{C}$onstructor *e.g.* $\mathcal{D} = \{app\}$ and $\mathcal{C} = \{a, cons, nil\}$
- Define $E^c_{\mathcal{C}}$ a set of contracting equations on $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$, such that $\mathcal{T}(\mathcal{C})^{\mathcal{S}}/_{=E^c_{\mathcal{C}}}$ is finite, *e.g.* $cons(x, cons(y, z)) = cons(x, z)$
- Define $E_{\mathcal{R}} = \{l = r \mid l \to r \in \mathcal{R}\}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$
- Theorem: If $\mathcal{R}$ is sufficiently complete then $\mathcal{T}(\mathcal{F})^{\mathcal{S}}/_{=E}$ is finite

$\mathcal{R}$ Sufficiently complete:

$\forall s \in \mathcal{T}(\mathcal{F})^{\mathcal{S}}. \exists t \in \mathcal{T}(\mathcal{C})^{\mathcal{S}}. s \to_{\mathcal{R}}^* t$



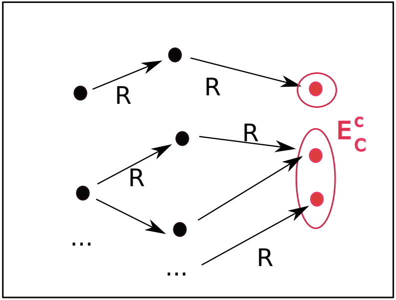
Equations guaranteeing termination of completion (III)

Proposed solution: use $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$

- $\mathcal{D}$efined and $\mathcal{C}$onstructor *e.g.* $\mathcal{D} = \{app\}$ and $\mathcal{C} = \{a, cons, nil\}$
- Define $E^c_{\mathcal{C}}$ a set of contracting equations on $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$, such that $\mathcal{T}(\mathcal{C})^{\mathcal{S}}/_{=E^c_{\mathcal{C}}}$ is finite, *e.g.* $cons(x, cons(y, z)) = cons(x, z)$
- Define $E_{\mathcal{R}} = \{l = r \mid l \to r \in \mathcal{R}\}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$
- Theorem: If $\mathcal{R}$ is sufficiently complete then $\mathcal{T}(\mathcal{F})^{\mathcal{S}}/_{=E}$ is finite

$\mathcal{R}$ Sufficiently complete:

$\forall s \in \mathcal{T}(\mathcal{F})^{\mathcal{S}}. \exists t \in \mathcal{T}(\mathcal{C})^{\mathcal{S}}. s \to_{\mathcal{R}}^{*} t$



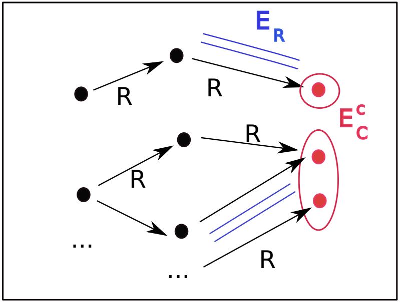
Equations guaranteeing termination of completion (III)

Proposed solution: use $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$

- $\mathcal{D}$efined and $\mathcal{C}$onstructor *e.g.* $\mathcal{D} = \{app\}$ and $\mathcal{C} = \{a, cons, nil\}$
- Define $E^c_{\mathcal{C}}$ a set of contracting equations on $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$, such that $\mathcal{T}(\mathcal{C})^{\mathcal{S}}/_{=E^c_{\mathcal{C}}}$ is finite, *e.g.* $cons(x, cons(y, z)) = cons(x, z)$
- Define $E_{\mathcal{R}} = \{l = r \mid l \to r \in \mathcal{R}\}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$
- Theorem: If $\mathcal{R}$ is sufficiently complete then $\mathcal{T}(\mathcal{F})^{\mathcal{S}}/_{=E}$ is finite

$\mathcal{R}$ Sufficiently complete:

$\forall s \in \mathcal{T}(\mathcal{F})^{\mathcal{S}}. \exists t \in \mathcal{T}(\mathcal{C})^{\mathcal{S}}. s \to_{\mathcal{R}}{}^{*} t$



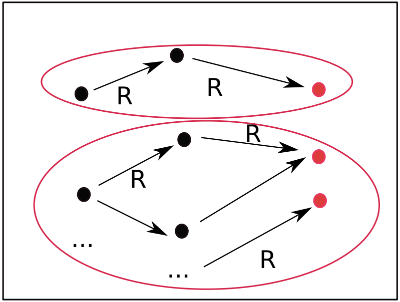
Equations guaranteeing termination of completion (III)

Proposed solution: use $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$

- $\mathcal{D}$efined and $\mathcal{C}$onstructor *e.g.* $\mathcal{D} = \{app\}$ and $\mathcal{C} = \{a, cons, nil\}$
- Define $E^c_{\mathcal{C}}$ a set of contracting equations on $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$, such that $\mathcal{T}(\mathcal{C})^{\mathcal{S}}/_{=E^c_{\mathcal{C}}}$ is finite, *e.g.* $cons(x, cons(y, z)) = cons(x, z)$
- Define $E_{\mathcal{R}} = \{l = r \mid l \to r \in \mathcal{R}\}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$
- Theorem: If $\mathcal{R}$ is sufficiently complete then $\mathcal{T}(\mathcal{F})^{\mathcal{S}}/_{=E}$ is finite

$\mathcal{R}$ Sufficiently complete:

$\forall s \in \mathcal{T}(\mathcal{F})^{\mathcal{S}}. \exists t \in \mathcal{T}(\mathcal{C})^{\mathcal{S}}. s \to_{\mathcal{R}}^* t$



Equations guaranteeing termination of completion (III)

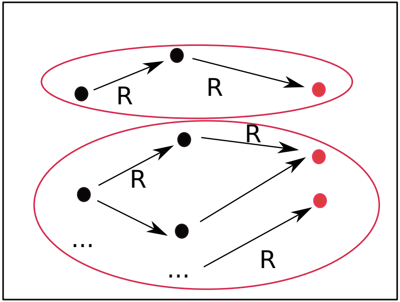
Proposed solution: use $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$

- $\mathcal{D}$efined and $\mathcal{C}$onstructor *e.g.* $\mathcal{D} = \{app\}$ and $\mathcal{C} = \{a, cons, nil\}$
- Define $E^c_{\mathcal{C}}$ a set of contracting equations on $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$, such that $\mathcal{T}(\mathcal{C})^{\mathcal{S}}/_{=E^c_{\mathcal{C}}}$ is finite, *e.g.* $cons(x, cons(y, z)) = cons(x, z)$
- Define $E_{\mathcal{R}} = \{l = r \mid l \to r \in \mathcal{R}\}$ and $E = E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$
- Theorem: If $\mathcal{R}$ is sufficiently complete then $\mathcal{T}(\mathcal{F})^{\mathcal{S}}/_{=E}$ is finite

$\mathcal{R}$ Sufficiently complete:

$\forall s \in \mathcal{T}(\mathcal{F})^{\mathcal{S}}.\exists t \in \mathcal{T}(\mathcal{C})^{\mathcal{S}}.s \to_{\mathcal{R}}{}^* t$

Demo: `demo_reverse.txt`



A word about Higher-Order functions

Static analysis of higher-order functional programs

- use higher-order formalisms: *e.g.* HORS [L. Ong, 2006], PMRS [L. Ong&S. Ramsay, 2011]

A word about Higher-Order functions

Static analysis of higher-order functional programs

- use higher-order formalisms: *e.g.* HORS [L. Ong, 2006], PMRS [L. Ong&S. Ramsay, 2011]
- use first-order formalisms (*e.g* tree automata and TRS) with an encoding of higher-order into first-order *e.g.* [N. Jones,1987]

A word about Higher-Order functions

Static analysis of higher-order functional programs

- use higher-order formalisms: *e.g.* HORS [L. Ong, 2006], PMRS [L. Ong&S. Ramsay, 2011]
- use first-order formalisms (*e.g* tree automata and TRS) with an encoding of higher-order into first-order *e.g.* [N. Jones,1987]

Example 4 (Encoding of H.O. functions into TRS)

Use an explicit function application operator '@'.

```
let rec map f l1 = match l1 with
  | [] -> []
  | h::t -> (f h) :: (map f t);;
```

becomes

$$@(@(map, f), nil) \to nil$$

$$@(@(map, f), cons(h, t)) \to cons(@(f, h), @(@(map, f), t))$$

A word about Higher-Order functions (II)

Is the @-encoding enough?

- Auhors of H.O. formalisms claim that the @-encoding is too imprecise

A word about Higher-Order functions (II)

Is the @-encoding enough?

- Auhors of H.O. formalisms claim that the @-encoding is too imprecise
- On H.O. examples of [L. Ong&S. Ramsay, 2011], we obtained similar results with the @-encoding, TRSs, and tree automata completion

A word about Higher-Order functions (II)

Is the @-encoding enough?

- Auhors of H.O. formalisms claim that the @-encoding is too imprecise
- On H.O. examples of [L. Ong&S. Ramsay, 2011], we obtained similar results with the @-encoding, TRSs, and tree automata completion

Example 5 (filter nz on any nat list, results in a list without 0)

```
let if2 c t e = match c with
  | true -> t
  | false -> e;;
```

```
let nz i= match i with
  | 0 -> false
  | S(x) -> true;;
```

```
let rec filter p l= match l with
 | [] -> []
 | h::t -> if2 (p h) (h::(filter p t)) (filter p t);;
```

A word about Higher-Order functions (II)

Is the @-encoding enough?

- Auhors of H.O. formalisms claim that the @-encoding is too imprecise
- On H.O. examples of [L. Ong&S. Ramsay, 2011], we obtained similar results with the @-encoding, TRSs, and tree automata completion

Example 5 (`filter` `nz` on any `nat` `list`, results in a list without 0)

```
let if2 c t e = match c with
  | true -> t
  | false -> e;;
```

```
let nz i= match i with
  | 0 -> false
  | S(x) -> true;;
```

```
let rec filter p l= match l with
  | [] -> []
  | h::t -> if2 (p h) (h::(filter p t)) (filter p t);;
```

Successful on some examples but needs to be investigated further!

A word about evaluation strategies

Example 6 (Terminating with call-by-need but not for call-by-value)

```
let rec sumList(x,y)= (x+y)::sumList(x+y,y+1);;
let rec nth i (x::l)= if i<=0 then x else nth (i-1) l;;
let sum x= nth x (sumList(0,0));;
```

A word about evaluation strategies

Example 6 (Terminating with call-by-need but not for call-by-value)

```
let rec sumList(x,y)= (x+y)::sumList(x+y,y+1);;
let rec nth i (x::l)= if i<=0 then x else nth (i-1) l;;
let sum x= nth x (sumList(0,0));;
```

$(sum\,4) = 10$ with call by need and diverges with call-by-value

A word about evaluation strategies

Example 6 (Terminating with call-by-need but not for call-by-value)

```
let rec sumList(x,y)= (x+y)::sumList(x+y,y+1);;
let rec nth i (x::l)= if i<=0 then x else nth (i-1) l;;
let sum x= nth x (sumList(0,0));;
```

$(sum\,4) = 10$ with call by need and diverges with call-by-value

Completion covers all reachable terms (for all strategies)

$\mathcal{R}^*((sum\,4)) \subseteq \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E})$ contains 10 (and intermediate computations)

A word about evaluation strategies

Example 6 (Terminating with call-by-need but not for call-by-value)

```
let rec sumList(x,y)= (x+y)::sumList(x+y,y+1);;
let rec nth i (x::l)= if i<=0 then x else nth (i-1) l;;
let sum x= nth x (sumList(0,0));;
```

$(sum\, 4) = 10$ with call by need and diverges with call-by-value

Completion covers all reachable terms (for all strategies)

$\mathcal{R}^*((sum\, 4)) \subseteq \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E})$ contains 10 (and intermediate computations)

Call-by-value $\leftrightarrow$ innermost strategy for TRSs

A word about evaluation strategies

Example 6 (Terminating with call-by-need but not for call-by-value)

```
let rec sumList(x,y)= (x+y)::sumList(x+y,y+1);;
let rec nth i (x::l)= if i<=0 then x else nth (i-1) l;;
let sum x= nth x (sumList(0,0));;
```

$(sum\,4) = 10$ with call by need and diverges with call-by-value

Completion covers all reachable terms (for all strategies)

$\mathcal{R}^*((sum\,4)) \subseteq \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E})$ contains 10 (and intermediate computations)

Call-by-value $\leftrightarrow$ innermost strategy for TRSs

Adapted tree automata completion for innermost strategy [with Y. Salmon]

A word about evaluation strategies

Example 6 (Terminating with call-by-need but not for call-by-value)

```
let rec sumList(x,y)= (x+y)::sumList(x+y,y+1);;
let rec nth i (x::l)= if i<=0 then x else nth (i-1) l;;
let sum x= nth x (sumList(0,0));;
```

$(sum\, 4) = 10$ with call by need and diverges with call-by-value

Completion covers all reachable terms (for all strategies)

$\mathcal{R}^*((sum\, 4)) \subseteq \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E})$ contains 10 (and intermediate computations)

Call-by-value $\leftrightarrow$ innermost strategy for TRSs

Adapted tree automata completion for innermost strategy [with Y. Salmon]

$\mathcal{R}^*_{in}((sum\, x)) \subseteq \mathcal{L}(\mathcal{A}^*_{\mathcal{R}_{in},E})$ contains no normal form (no result)

A word about built-in types

Recall this example:

Example 7 (`filter nz` on any `nat list`, results in a list without 0)

```
let if2 c t e = match c with
  | true -> t
  | false -> e;;
```

```
let nz i= match i with
  | 0 -> false
  | S(x) -> true;;
```

Programs usually use machine integers instead of Peano numbers

A word about built-in types

Recall this example:

Example 7 (`filter nz` on any `nat list`, results in a list without `0`)

```
let if2 c t e = match c with
  | true -> t
  | false -> e;;
```

```
let nz i= match i with
  | 0 -> false
  | S(x) -> true;;
```

Programs usually use machine integers instead of Peano numbers

Lattice Tree Automata completion [with Legay, Le Gall, Murat, 2013]

LTA completion permits to seamlessly plug abstract domains in ARTMC

A word about built-in types

Recall this example:

Example 7 (`filter nz` on any `nat list`, results in a list without `0`)

```
let if2 c t e = match c with
  | true -> t
  | false -> e;;
```

```
let nz i= match i with
  | 0 -> false
  | S(x) -> true;;
```

Programs usually use machine integers instead of Peano numbers

Lattice Tree Automata completion [with Legay, Le Gall, Murat, 2013]

LTA completion permits to seamlessly plug abstract domains in ARTMC

e.g. integer lists with no zero:

$cons(q_i, q_l) \rightarrow q_l$	$[-\infty; -1] \rightarrow q_i$
$nil \rightarrow q_l$	$[1; +\infty] \rightarrow q_i$

What about the presentation of the results/annotations?

A simple automaton for the *A* then *B* lists

```
Automaton A0
States qA, qB, qnil, qlB, qlAB
Final States qlAB
Transitions
  A -> qA
  B -> qB
  nil -> qnil
  cons(qB,qnil) -> qlB
  cons(qB,qlB) -> qlB
  cons(qA,qlB) -> qlAB
  cons(qA, qlAB) -> qlAB
```

Any suggestion for a short textual/graphical format is welcome!

What about the presentation of the results/annotations?

Contracts [D. Xu, 2009]

```
contract rev = {l | ab l} -> {l | ba l};;
```

where `ab` and `ba` are user defined functions discriminating the ≪A then B lists≫ etc. Contracts can be dynamically or statically checked.

What about the presentation of the results/annotations?

Contracts [D. Xu, 2009]

```
contract rev = {l | ab l} -> {l | ba l};;
```

where ab and ba are user defined functions discriminating the ≪A then B lists≫ etc. Contracts can be dynamically or statically checked.

Liquid Types (and variants) [N. Vazou, P. Rondon, R. Jhala, 2013]

```
rev :: [a]<{\h v -> h <= v}> -> [a]<{\h v -> h >= v}
```

Liquid types are statically checked.

What about the presentation of the results/annotations?

Contracts [D. Xu, 2009]

```
contract rev = {l | ab l} -> {l | ba l};;
```

where `ab` and `ba` are user defined functions discriminating the ≪A then B lists≫ etc. Contracts can be dynamically or statically checked.

Liquid Types (and variants) [N. Vazou, P. Rondon, R. Jhala, 2013]

```
rev :: [a]<{\h v -> h <= v}> -> [a]<{\h v -> h >= v}
```

Liquid types are statically checked.

Two remarks and one question

+ Those techniques prove stronger properties (*e.g.* `quicksort` sorts)

What about the presentation of the results/annotations?

Contracts [D. Xu, 2009]

```
contract rev = {l | ab l} -> {l | ba l};;
```

where `ab` and `ba` are user defined functions discriminating the ≪A then B lists≫ etc. Contracts can be dynamically or statically checked.

Liquid Types (and variants) [N. Vazou, P. Rondon, R. Jhala, 2013]

```
rev :: [a]<{\h v -> h <= v}> -> [a]<{\h v -> h >= v}
```

Liquid types are statically checked.

Two remarks and one question

+ Those techniques prove stronger properties (*e.g.* `quicksort` sorts)
– (Co)-Domains annotations are given by the user (we infer them)

What about the presentation of the results/annotations?

Contracts [D. Xu, 2009]

```
contract rev = {l | ab l} -> {l | ba l};;
```

where `ab` and `ba` are user defined functions discriminating the ≪A then B lists≫ etc. Contracts can be dynamically or statically checked.

Liquid Types (and variants) [N. Vazou, P. Rondon, R. Jhala, 2013]

```
rev :: [a]<{\h v -> h <= v}> -> [a]<{\h v -> h >= v}
```

Liquid types are statically checked.

Two remarks and one question

+ Those techniques prove stronger properties (*e.g.* `quicksort` sorts)
– (Co)-Domains annotations are given by the user (we infer them)
• Can we define user friendly "language annotations" close to types?

Conclusion

1. Define equations guaranteeing termination of completion ✓ ⚠
2. Deal with higher order functions ⚠
3. Take evaluation strategies into account
    - call by value (*e.g.* Ocaml) $\approx$ innermost rewrite strategy ✓
    - call by need (*e.g.* Haskell) $\approx$ outermost rewrite strategy $+$ sharing
    - order in pattern matching $\approx$ priority rewrite strategy
4. Deal with built-in types ✓
5. Modularity of the analysis
6. User friendly way to display/define language annotations ...

Further research

- Find a translation from OCaml to TRS s.t.
  - Typing is preserved
  - Higher-order functions can be encoded
  - OCaml pattern matching exhaustivity $\Rightarrow$ TRS sufficient completeness

Further research

- Find a translation from OCaml to TRS s.t.
  - Typing is preserved
  - Higher-order functions can be encoded
  - OCaml pattern matching exhaustivity $\Rightarrow$ TRS sufficient completeness
- Find other criteria guaranteeing finiteness of $\mathcal{T}(\mathcal{F})/_{=_E}$ or $\mathcal{T}(\mathcal{C})/_{=_E}$

Further research

- Find a translation from OCaml to TRS s.t.
  - Typing is preserved
  - Higher-order functions can be encoded
  - OCaml pattern matching exhaustivity $\Rightarrow$ TRS sufficient completeness
- Find other criteria guaranteeing finiteness of $\mathcal{T}(\mathcal{F})/_{=_E}$ or $\mathcal{T}(\mathcal{C})/_{=_E}$

  e.g. Discard the "sufficient completeness" requirement

Example 8 (sumList is not sufficiently complete)

```
let rec sumList(x,y)= (x+y)::sumList(x+y,y+1);;
let rec nth i (x::l)= if i<=0 then x else nth (i-1) l;;
let sum x= nth x (sumList(0,0));;
```

Completion algorithm

Tree automata completion principle

1. complete $\mathcal{A}$ with new transitions into $\mathcal{A}^1_{\mathcal{R}}, \mathcal{A}^2_{\mathcal{R}}, \ldots$

Completion algorithm

Tree automata completion principle

1. complete $\mathcal{A}$ with new transitions into $\mathcal{A}_{\mathcal{R}}^1, \mathcal{A}_{\mathcal{R}}^2, \ldots$
   $\forall l \to r \in \mathcal{R}$, $\forall q \in \mathcal{Q}$, $\forall \sigma : \mathcal{X} \mapsto \mathcal{Q}$:

$$\begin{array}{ccc} l\sigma & \xrightarrow[\mathcal{R}]{} & r\sigma \\ {\scriptstyle \mathcal{A}_{\mathcal{R}}^i}\big\downarrow{\scriptstyle *} & & \\ q & & \end{array}$$

Completion algorithm

Tree automata completion principle

1. complete $\mathcal{A}$ with new transitions into $\mathcal{A}_{\mathcal{R}}^1, \mathcal{A}_{\mathcal{R}}^2, \ldots$
   $\forall l \to r \in \mathcal{R}$, $\forall q \in \mathcal{Q}$, $\forall \sigma : \mathcal{X} \mapsto \mathcal{Q}$:

$$\begin{tikzcd} l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}_{\mathcal{R}}^i"', "*"] & r\sigma \arrow[d, "*"', "\mathcal{A}_{\mathcal{R}}^{i+1}"] \\ q & q' \arrow[l, "\mathcal{A}_{i+1}"] \end{tikzcd}$$

Completion algorithm

Tree automata completion principle

1. complete $\mathcal{A}$ with new transitions into $\mathcal{A}^1_{\mathcal{R}}, \mathcal{A}^2_{\mathcal{R}}, \ldots$
   $\forall l \to r \in \mathcal{R}$, $\forall q \in \mathcal{Q}$, $\forall \sigma : \mathcal{X} \mapsto \mathcal{Q}$:

$$
\begin{tikzcd}
l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_{\mathcal{R}}"', "*"] & r\sigma \arrow[d, "*"', "\mathcal{A}^{i+1}_{\mathcal{R}}"] \\
q & q' \arrow[l, "\mathcal{A}_{i+1}"]
\end{tikzcd}
$$

2. use approximation equations of E to (possibly) converge on $\mathcal{A}^*_{\mathcal{R},E}$

Completion algorithm (II)

Definition 9 (Set $E^c_{\mathcal{C}}$ of contracting equations)

The set of well-sorted equations $E^c_{\mathcal{C}}$ is *contracting* if its equations are of the form $u = u|_p$ with u linear and $p \neq \Lambda$ and if the set of normal forms of $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$ w.r.t. the TRS $\overrightarrow{E^c_{\mathcal{C}}} = \{u \to v \mid u = v \in E^c_{\mathcal{C}}\}$ is finite.

R/E-coherence

Languages recognized by states of $\mathcal{A}$ (ϵ-free) are E-equivalent terms.