Program Transformation for Reliable Real Numbers

Pierre NERON

École polytechnique - INRIA

October 4, 2013

Real Numbers

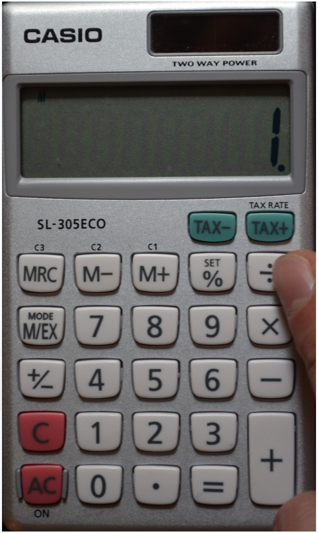
$$(1/3) \times 3 = ?$$

Real Numbers

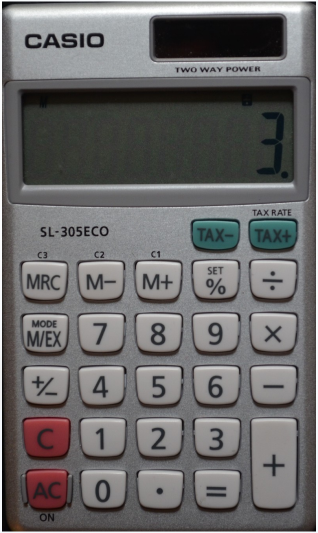
$$(1/3) \times 3 = 1$$

(Un)Reliable Real Numbers ?

(Un)Reliable Real Numbers ?

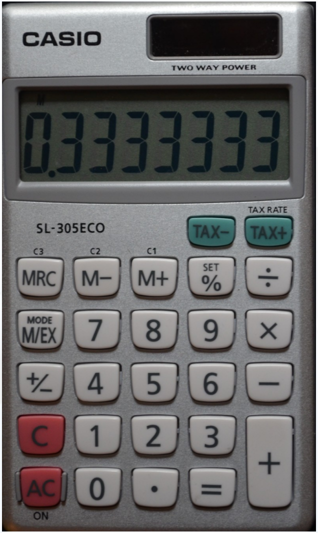


(Un)Reliable Real Numbers ?

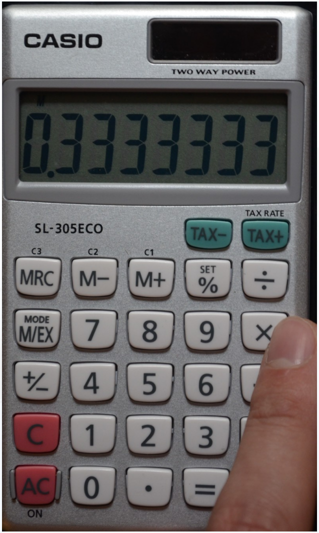


(Un)Reliable Real Numbers ?

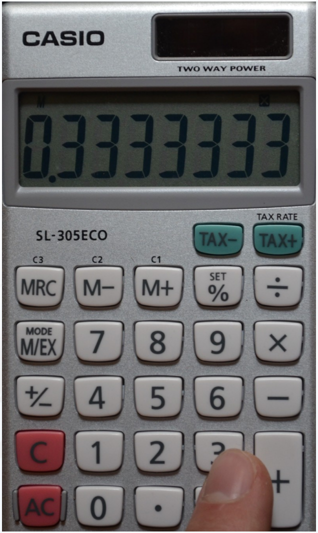
(Un)Reliable Real Numbers ?



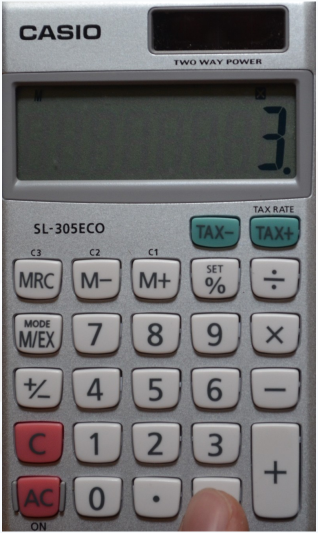
(Un)Reliable Real Numbers ?



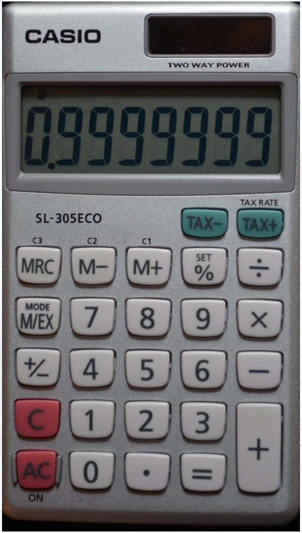
(Un)Reliable Real Numbers ?



(Un)Reliable Real Numbers ?



(Un)Reliable Real Numbers ?



Floating Point Numbers

```
        Objective Caml version 3.12.1

#
```

Floating Point Numbers

```
        Objective Caml version 3.12.1

#   0.2 +. 0.1;;
```

Floating Point Numbers

```
        Objective Caml version 3.12.1

#   0.2 +. 0.1;;
- : float = 0.300000000000000044

#
```

Floating Point Numbers

```
        Objective Caml version 3.12.1

#   0.2 +. 0.1;;
- : float = 0.300000000000000044

#   0.1 +. 0.2 = 0.15 +. 0.15;;
```

Floating Point Numbers

```
        Objective Caml version 3.12.1

#   0.2 +. 0.1;;
- : float = 0.300000000000000044

#   0.1 +. 0.2 = 0.15 +. 0.15;;
- : bool = false

#
```

Floating Point Numbers

```
          Objective Caml version 3.12.1

#   0.2 +. 0.1;;
- : float = 0.300000000000000044

#   0.1 +. 0.2 = 0.15 +. 0.15;;
- : bool = false

#   (sqrt 2.) *. (sqrt 2.) > 2.;;
```

Floating Point Numbers

```
        Objective Caml version 3.12.1

#   0.2 +. 0.1;;
- : float = 0.300000000000000044

#   0.1 +. 0.2 = 0.15 +. 0.15;;
- : bool = false

#   (sqrt 2.) *. (sqrt 2.) > 2.;;
- : bool = true
```

In programs

```
        Objective Caml version 3.12.1

#
```

In programs

```
        Objective Caml version 3.12.1

# if 0.1 +. 0.2 = 0.15 +. 0.15
  then 1000
  else 0;;
```

In programs

```
        Objective Caml version 3.12.1

# if 0.1 +. 0.2 = 0.15 +. 0.15
  then 1000
  else 0;;
- : int = 0

#
```

In programs

```
        Objective Caml version 3.12.1

# if 0.1 +. 0.2 = 0.15 +. 0.15
  then 1000
  else 0;;
- : int = 0

# if (sqrt 2.) *. (sqrt 2.) > 2.
  then print_string "turn_right"
  else print_string "turn_left";;
```

In programs

```
        Objective Caml version 3.12.1

# if 0.1 +. 0.2 = 0.15 +. 0.15
  then 1000
  else 0;;
- : int = 0

# if (sqrt 2.) *. (sqrt 2.) > 2.
  then print_string "turn_right"
  else print_string "turn_left";;
turn right
- : unit = ()
```

Infinite vs. Finite

- Infinite behaviors:

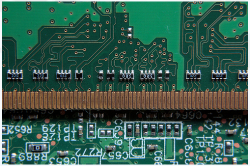
$$\sqrt{2} = 1.4142135623... \qquad 1/7 = 0.1428571428...$$

Infinite vs. Finite

- Infinite behaviors:

$$\sqrt{2} = 1.4142135623... \qquad 1/7 = 0.1428571428...$$

- In a finite world:



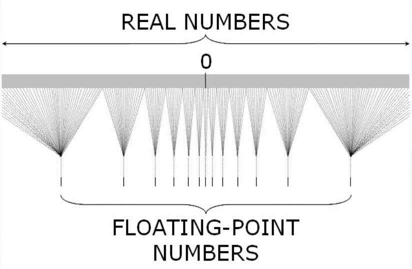
$\sqrt{}$ and $\div$ Elimination

"There is nothing (right well beloved Students in the Mathematickes) that is so troublesome to Mathematicall practice, not that doth molest and hinder Calculators, then the Multiplications, Divisions, square and cubical Extraction of great numbers, which besides the tedious expence of time, are for the most part subject to many slippery errors."

John Napier 1614

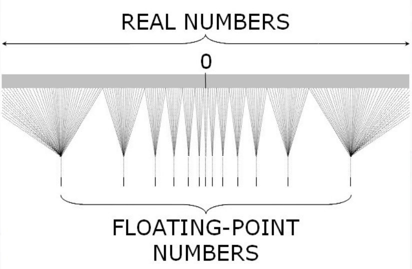
Finite representations

- Rounding with finite representations:



Finite representations

- Rounding with finite representations:



- can provoke ERRORS:

$$\sqrt{2} \times \sqrt{2} > 2$$

Reliable Real numbers

- Working on the representation
  - Static analysis
  - Abstract Interpretation
  - Program transformation (Precision)
  - Reasoning about $\mathbb{F}$
- Changing representation
  - Interval arithmetic $(=,>)$
  - Algebraic numbers (ES)
  - Lazy evaluation (ES)

Program Transformation ?

Program Transformation ?

- A program that manipulates programs ...

Program Transformation ?

- A program that manipulates programs ...

```
foo
-----------------------------
let x = read(input)
in sqrt(x)*sqrt(x) > 2
```

Program Transformation ?

- A program that manipulates programs ...

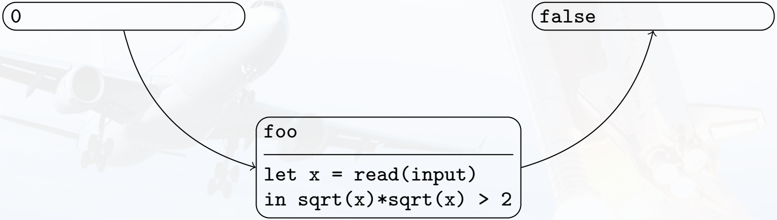
0

foo

```
let x = read(input)
in sqrt(x)*sqrt(x) > 2
```

Program Transformation ?

- A program that manipulates programs ...



Program Transformation ?

- A program that manipulates programs ...

```
foo
------------------------------
let x = read(input)
in sqrt(x)*sqrt(x) > 2
```

Program Transformation ?

- A program that manipulates programs ...

```
foo
-----------------------
let x = read(input)
in sqrt(x)*sqrt(x) > 2
```

```
transfo
-----------------------
p:= read(input);
o:= transform(p);
print(o)
```

Program Transformation ?

- A program that manipulates programs ...

```
foo
----------------------------
let x = read(input)
in sqrt(x)*sqrt(x) > 2
```

```
transfo
----------------------------
p:= read(input);
o:= transform(p);
print(o)
```

Program Transformation ?

- A program that manipulates programs ...

foo

```
let x = read(input)
in sqrt(x)*sqrt(x) > 2
```

transfo

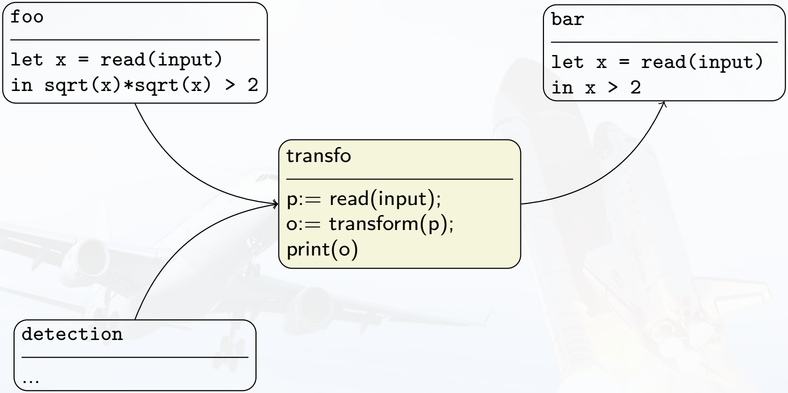
```
p:= read(input);
o:= transform(p);
print(o)
```

bar

```
let x = read(input)
in x > 2
```

Program Transformation ?

- A program that manipulates programs ...



Program Transformation ?

- A program that manipulates programs ...

```
foo
--------------------------
let x = read(input)
in sqrt(x)*sqrt(x) > 2
```

```
bar
--------------------------
let x = read(input)
in x > 2
```

```
transfo
--------------------------
p:= read(input);
o:= transform(p);
print(o)
```

```
detection
--------------------------
...
```

```
detection_sqfree
--------------------------
...
```

Objectives

- Define a program transformation that removes $\sqrt{\ }$ and $/$
- Exact computation with other operations $+, -, \times$
- Embedded programs context:
  - subset of languages
  - program does not fail $(1/0; \sqrt{-1})$
  - fixed size data structures
  - proving the transformation using the Program Verification System (PVS)

$/$ and $\sqrt{\ }$ elimination

Constrained Anti-unification

Certification with a proof assistant

/ and $\sqrt{\ }$ elimination

- Language
- Elim $\mathbb{B}$
- Variable definition

Constrained Anti-unification

Certification with a proof assistant

Straight line programs

$+, -, \times, /, \sqrt{}$

$\wedge, \vee, \neg$

$=, \neq, >, \geq \ldots$

let x = ... in ...

(...,...), fst(...), snd(...)

if ... then ... else

Exact computation with $+, -, \times$

- $/$ or $\sqrt{}$ require an infinite number of digits to be exactly computed:

$$1/3 = 0.333... \qquad\qquad \sqrt{2} = 1.414...$$

Exact computation with $+, -, \times$

- $/$ or $\sqrt{}$ require an infinite number of digits to be exactly computed:

$$1/3 = 0.333... \qquad \sqrt{2} = 1.414...$$

- $+, -, \times$ can always be exactly computed with a finite memory:

$$1.02 \times 50.02 = 51.0204$$

Exact computation with $+, -, \times$

- $/$ or $\sqrt{}$ require an infinite number of digits to be exactly computed:

$$1/3 = 0.333... \qquad \sqrt{2} = 1.414...$$

- $+, -, \times$ can always be exactly computed with a finite memory:

$$1.02 \times 50.02 = 51.0204$$

No loops or recursion $\Longrightarrow$ static analysis can do it

Specification

Formal Specification

Given $[\![p]\!]_{Env}$, the real number semantics of p in an environment Env:

$$\forall\ \mathrm{p} \in \mathrm{P}, \forall\ Env,\ [\![\ \mathrm{p}\ ]\!]_{Env}\ \neq Fail\ \Rightarrow$$
$$[\![\ Elim(\mathrm{p})\ ]\!]_{Env}\ =\ [\![\ \mathrm{p}\ ]\!]_{Env}\ \wedge\ Elim(\mathrm{p})\ \in\ \mathrm{P}_{\mathrm{N}_{\surd,/}}$$

$\mathrm{P}_{\mathrm{N}_{\surd,/}}$: control flow does not depend on $\sqrt{\ }$ and $/$

Specification

Formal Specification

Given $[\![p]\!]_{Env}$, the real number semantics of p in an environment Env:

$$\forall\ \mathrm{p} \in \mathrm{P}, \forall\ Env,\ [\![\ \mathrm{p}\ ]\!]_{Env}\ \neq Fail\ \Rightarrow$$
$$[\![\ Elim(\mathrm{p})\ ]\!]_{Env}\ =\ [\![\ \mathrm{p}\ ]\!]_{Env}\ \wedge\ Elim(\mathrm{p})\ \in\ \mathrm{P}_{\mathrm{N}_{\sqrt{},/}}$$

$\mathrm{P}_{\mathrm{N}_{\sqrt{},/}}$: control flow does not depend on $\sqrt{}$ and $/$

Keep the size of the produced code $Elim$(p) reasonable

$/$ and $\sqrt{}$ elimination

Language

Elim $\mathbb{B}$

Variable definition

Constrained Anti-unification

Certification with a proof assistant

Elimination in $\mathbb{B}$

- Quantifier elimination

Example

$$\sqrt{a+\sqrt{b}} > c$$

Elimination in $\mathbb{B}$

- Quantifier elimination

Example

$$\sqrt{a + \sqrt{b}} > c$$

$$\vdots$$

$$\forall\ x,\ \forall\ y,\ \ y \geq 0 \wedge y^2 = b \wedge x \geq 0 \wedge x^2 = y + a \wedge x > c$$

Elimination in $\mathbb{B}$

- Quantifier elimination

Example

$$\sqrt{a + \sqrt{b}} > c$$

$$\vdots$$

$$\forall\ x,\ \forall\ y,\ y \geq 0 \land y^2 = b \land x \geq 0 \land x^2 = y + a \land x > c$$

$$\vdots$$

Too many free variables

Elimination in $\mathbb{B}$

Example

$$\sqrt{a+\sqrt{b}} > c$$

Elimination in $\mathbb{B}$

Example

$$\sqrt{a + \sqrt{b}} > c$$

$$\vdots$$

$$a + \sqrt{b} > c^2 \qquad \vee \qquad c < 0$$

Elimination in $\mathbb{B}$

Example

$$\sqrt{a + \sqrt{b}} > c$$

$$\vdots$$

$$a + \sqrt{b} > c^2 \qquad \vee \qquad c < 0$$

$$\vdots$$

$$b > (c^2 - a)^2 \qquad \vee \qquad c^2 - a < 0 \qquad \vee \qquad c < 0$$

$/$ and $\sqrt{\ }$ elimination in $\mathbb{B}$

Recursive algorithm:

- Reduction to one head division
- Division elimination:

$$\frac{A}{B} \geq \frac{C}{D} \longrightarrow A.B.D^2 - C.D.B^2 \geq 0$$

$/$ and $\sqrt{\ }$ elimination in $\mathbb{B}$

Recursive algorithm:

- Reduction to one head division
- Division elimination:

$$\frac{A}{B} \geq \frac{C}{D} \longrightarrow A.B.D^2 - C.D.B^2 \geq 0$$

- Factorization with one square root
- Square root elimination:

$$P.\sqrt{Q} + R > 0 \longrightarrow$$
$$(P > 0 \wedge R > 0) \vee$$
$$(P > 0 \wedge P^2.Q - R^2 > 0) \vee$$
$$(R > 0 \wedge P^2.Q - R^2 < 0)$$

$/$ and $\sqrt{}$ elimination in $\mathbb{B}$

Recursive algorithm:

- Reduction to one head division
- Division elimination:

$$\frac{A}{B} \geq \frac{C}{D} \longrightarrow A.B.D^2 - C.D.B^2 \geq 0$$

- Factorization with one square root
- Square root elimination:

$$\begin{aligned} & P.\sqrt{Q} + R > 0 \longrightarrow \\ & \quad \text{let } (at_p, at_r, at_{pr}, at_{epr}) = \\ & \qquad (P > 0, R > 0, P^2.Q - R^2 > 0, P^2.Q - R^2 \neq 0) \\ & \quad \text{in } (at_p \wedge at_r) \vee (at_p \wedge at_{pr}) \vee (at_r \wedge \neg at_{pr} \wedge at_{epr}) \end{aligned}$$

$/$ and $\sqrt{}$ elimination in $\mathbb{B}$

Recursive algorithm:

- Reduction to one head division
- Division elimination:

$$\frac{A}{B} \geq \frac{C}{D} \longrightarrow A.B.D^2 - C.D.B^2 \geq 0$$

- Factorization with one square root
- Square root elimination:

$$P.\sqrt{Q} + R > 0 \longrightarrow$$

$$\text{let } (\text{at}_p, \text{at}_r, \text{at}_{pr}, \text{at}_{epr}) = \\ (P > 0, R > 0, P^2.Q - R^2 > 0, P^2.Q - R^2 \neq 0) \\ \text{in } (\text{at}_p \wedge \text{at}_r) \vee (\text{at}_p \wedge \text{at}_{pr}) \vee (\text{at}_r \wedge \neg\text{at}_{pr} \wedge \text{at}_{epr})$$

Complexity: each atom produces $4^{\#(\sqrt{})}$ atoms max

$/$ and $\sqrt{}$ elimination

Language

Elim $\mathbb{B}$

Variable definition

Constrained Anti-unification

Certification with a proof assistant

Variable Definition

Example

$$\text{let } x = a + \sqrt{b + c} \text{ in } x > 0$$

Variable Definition

Example

$$\text{let } x = a + \sqrt{b + c} \text{ in } x > 0$$

$$\vdots$$

$$a + \sqrt{b + c} > 0$$

Variable Definition

Example

$$\text{let } x = a + \sqrt{b + c} \text{ in } x > 0$$

$$\vdots$$

$$a + \sqrt{b + c} > 0$$

Output code size explosion

Variable Definition

Example

$$\text{let } x = a + \sqrt{b + c} \text{ in } x > 0$$

$$\vdots$$

$$\text{let } (x_1, x_2) = (a, b + c) \text{ in } x_1 + \sqrt{x_2} > 0$$

Test in variables

Example

let x =
  if F
    then $a_1 + \sqrt{a_2}$
    else $\frac{b_1}{b_2}$
  in P

$\longrightarrow$

let (x_1,x_2,x_3) =
  if F
  then
   $(a_1, a_2, 1)$
  else
   $(b_1, 0, b_2)$
 in P[x:= $\frac{x_1+\sqrt{x_2}}{x_3}$]

Test in variables

Example

let x =
  if F
   then $a_1 + \sqrt{a_2}$
   else $\frac{b_1}{b_2}$
  in P

$\longrightarrow$

let (x_1,x_2,x_3) =
  if F
  then
   $(a_1, a_2, 1)$
  else
   $(b_1, 0, b_2)$
 in P[x:= $\frac{x_1+\sqrt{x_2}}{x_3}$]

Expressions anti-unification

$a_1 + \sqrt{a_2}$

$\frac{b_1}{b_2}$

$\longrightarrow$ $\frac{x_1+\sqrt{x_2}}{x_3}$

$[x_1 := a_1,\ x_2 := a_2,\ x_3 := 1]$

$[x_1 := b_1,\ x_2 := 0,\ x_3 := b_2]$

Elim_{let}

$$\text{let x} = \text{cases}(e_1, ..., e_m) \text{ in P}$$

$$\longrightarrow$$

$$\forall\ i,\ \ e_i = T[x_1 \mapsto se_{i1}, ..., x_n \mapsto \mathrm{se}_{in}]$$

$$\longrightarrow$$

$$\begin{aligned}&\text{let } (x_1, ..., x_n) = \\ &\qquad \text{cases}((se_{11}, ..., se_{1n}), ..., (se_{m1}, ..., se_{mn})) \\ &\text{in } \mathrm{P}[\mathrm{x} \mapsto T]\end{aligned}$$

Program Classes Equivalence

Theorem (Prog is semantically equivalent to $P_{N_{\sqrt{},/}}$)

$$\forall\ p \in Prog,\ \exists\ p_{sq} \in P_{N_{\sqrt{},/}},\ \forall Env,$$

$$[\![p]\!]_{Env} \neq Fail \implies [\![p_{sq}]\!]_{Env} = [\![p]\!]_{Env}$$

Corollary

A program that computes a value of type $\mathbb{B}^n$ is semantically equivalent to a square root and division free program.

Therefore this program can be exactly computed

$/$ and $\sqrt{\ }$ elimination

Constrained Anti-unification

- Definition
- Functions

Certification with a proof assistant

Anti-unification

Dual of the unification problem:

Example

Given:

$$f(g(a), b, h(c)) \qquad f(g(a'), h(b'), c')$$

Anti-unification

Dual of the unification problem:

Example

Given:

$$f(g(a), b, h(c)) \qquad f(g(a'), h(b'), c')$$

We can compute the following template:

$$f(g(x), y, z)$$

Anti-unification modulo arithmetic

Example

Given:

$$\frac{a+b}{c \cdot d} \qquad \sqrt{a' + b'} + c' \cdot e'$$

Anti-unification modulo arithmetic

Example

Given:

$$\frac{a+b}{c \cdot d} \qquad \sqrt{a' + b'} + c' \cdot e'$$

We can compute the following template:

$$\frac{x + \sqrt{y}}{z}$$

Anti-unification modulo arithmetic

Example

Given:

$$\frac{a+b}{c \cdot d} \qquad\qquad \sqrt{a' + b'} + c' \cdot e'$$

We can compute the following template:

$$\frac{x + \sqrt{y}}{z}$$

$$\frac{a+b}{c \cdot d} = \frac{(a+b)+\sqrt{0}}{c \cdot d} \qquad\qquad \sqrt{a' + b'} + c' \cdot e' = \frac{c' \cdot e' + \sqrt{a'+b'}}{1}$$

Constrained anti-unification

Definition ($\sqrt{}, /$-constrained template)

Given a set of arithmetic terms $\mathcal{S} \in \mathcal{A}$, a term $t \in \mathcal{A}$ is a $\sqrt{}, /$-constrained template of $\mathcal{S}$ when:

$$\forall s \in \mathcal{S},\ \exists\ \sigma$$

$$t\sigma \ = \ s \qquad \wedge \qquad \mathcal{I}(\sigma) \subset \mathcal{A} \setminus \{\sqrt{}, /\}$$

where $\mathcal{I}(\sigma)$ is the image of σ, *i.e.*, $\{t \mid \exists x,\ \sigma(x) = t\}$

Properties

Proposition ($\sqrt{}$, /-constrained anti-unification is complete)

Given a set $\mathcal{S}$ of arithmetic expressions, we can always find a constrained template for $\mathcal{S}$

Properties

Proposition ($\sqrt{}$, /-constrained anti-unification is complete)

Given a set $\mathcal{S}$ of arithmetic expressions, we can always find a constrained template for $\mathcal{S}$

Proof.

when $\mathcal{S} = \{s_1, s_2\}$: $x \times s_1 + (1 - x) \times s_2$ $\square$

Properties

Proposition ($\sqrt{}$, /-constrained anti-unification is complete)

Given a set $\mathcal{S}$ of arithmetic expressions, we can always find a constrained template for $\mathcal{S}$

Proof.

when $\mathcal{S} = \{s_1, s_2\}$: $x \times s_1 + (1 - x) \times s_2$ $\square$

However : size of this term bigger than sum of sizes of the inputs

Properties

Proposition ($\sqrt{}$, $/$-constrained anti-unification is complete)

Given a set $\mathcal{S}$ of arithmetic expressions, we can always find a constrained template for $\mathcal{S}$

Proof.

when $\mathcal{S} = \{s_1, s_2\}$: $x \times s_1 + (1 - x) \times s_2$ $\square$

However : size of this term bigger than sum of sizes of the inputs

Example

$$\{\sqrt{2},\quad 2+\sqrt{3}\} \quad \longrightarrow \quad x \times \sqrt{2} + (1-x) \times (2+\sqrt{3})$$

Properties

Proposition ($\sqrt{}$, $/$-constrained anti-unification is complete)

Given a set $\mathcal{S}$ of arithmetic expressions, we can always find a constrained template for $\mathcal{S}$

Proof.

when $\mathcal{S} = \{s_1, s_2\}$: $x \times s_1 + (1 - x) \times s_2$ $\square$

However : size of this term bigger than sum of sizes of the inputs

Example

$$\{\sqrt{2}, \quad 2 + \sqrt{3}\} \quad \longrightarrow \quad x + \sqrt{y}$$

Algorithm

- uses dag representation to minimize number of $\sqrt{\ }$

  $Elim_{\mathbb{B}}$ produces $4^{\#(\sqrt{})}$ atoms max

- relies on the following canonical form of arithmetic expressions:

$$\frac{\sum_{i=1}^{n} a_i \prod_{j_i=1}^{m_i} \sqrt{b_{j_i}}}{\sum_{i=1}^{n} c_i \prod_{j_i=1}^{m_i} \sqrt{d_{j_i}}}$$

- we can find a template such that $|t|_{\sqrt{}} = max_{s\in\mathcal{S}}(|s|_{\sqrt{}})$

$/$ and $\sqrt{}$ elimination

Constrained Anti-unification

- Definition
- Functions

Certification with a proof assistant

Functions

- Real programs contain function definitions and applications
- Inline $\Longrightarrow$ Size increases and lost of the program structure
- Transform functions directly

Function transformation

- Still relies on two constrained anti-unification

Example

let f x $= 3\text{x} + \sqrt{a}$ in ... $\text{f}(b)$...$\text{f}(c + d\sqrt{e})$ ...

Function transformation

- Still relies on two constrained anti-unification

Example

let f x = 3x + $\sqrt{a}$ in ... f(b)...f($c + d\sqrt{e}$) ...

Input transformation $\longrightarrow$

let f x_1 x_2 x_3 $= 3.(x_1 + x_2\sqrt{x_3}) + \sqrt{a}$ in ... f($b, 0, 0$)... f(c, d, e)...

Function transformation

- Still relies on two constrained anti-unification

Example

let f x $= 3x + \sqrt{a}$ in ... f(b)...f($c + d\sqrt{e}$) ...

Input transformation $\longrightarrow$

let f $x_1\ x_2\ x_3 = 3.(x_1 + x_2\sqrt{x_3}) + \sqrt{a}$ in ... f$(b, 0, 0)$... f(c, d, e)...

Output transformation $\longrightarrow$

let f $x_1\ x_2\ x_3 = (3.x_1, 3x_2, x_3, a)$ in

... let $(y_1, y_2, y_3, y_4) =$ f$(b, 0, 0)$ in $y_1 + y_2\sqrt{y_3} + \sqrt{y_4}$

... let $(y_1, y_2, y_3, y_4) =$ f(c, d, e) in $y_1 + y_2\sqrt{y_3} + \sqrt{y_4}$

Function input (arguments)

$$\text{let f x} = \text{body in} \ldots \text{f(e)} \ldots \text{f(e')}$$

$$\longrightarrow$$

$$\text{e} = \text{t}[\text{x}_1 \mapsto \text{e}_1,\ldots,\text{x}_n \mapsto \text{e}_n] \qquad \text{e'} = \text{t}[\text{x}_1 \mapsto \text{e'}_1,\ldots,\text{x}_n \mapsto \text{e'}_n]$$

$$\longrightarrow$$

$$\text{let f x}_1 \ldots \text{x}_n = \text{body}[\text{x} \mapsto \text{t}] \text{ in} \ldots \text{f}(\text{e}_1,\ldots,\text{e}_n) \ldots \text{f}(\text{e'}_1,\ldots,\text{e'}_n)$$

Function input (arguments)

$$\text{let f x} = \text{body in} \dots \text{f(e)} \dots \text{f(e')}$$

$$\longrightarrow$$

$$e = t[x_1 \mapsto e_1, \dots, x_n \mapsto e_n] \qquad e' = t[x_1 \mapsto e'_1, \dots, x_n \mapsto e'_n]$$

$$\longrightarrow$$

$$\text{let f } x_1 \dots x_n = \text{body}[x \mapsto t] \text{ in} \dots f(e_1,\dots,e_n) \dots f(e'_1,\dots,e'_n)$$

$\Longrightarrow$ No more square root or division in the function calls

Function output

let f x = if ... e ... e'... in scope

$$\longrightarrow$$

$e = t[x_1 \mapsto e_1,\ldots,x_n \mapsto e_n]$ $\qquad$ $e' = t[x_1 \mapsto e'_1,\ldots,x_n \mapsto e'_n]$

$$\longrightarrow$$

let f x = if ... $(e_1,\ldots,e_n)$ ... $(e'_1,\ldots,e'_n)$...
in scope[f(a) $\mapsto$ let x_1 ... x_n = f(a) in t]

Function output

$$\text{let f x} = \text{if} \ldots \text{ e } \ldots \text{ e'} \ldots \text{ in scope}$$

$$\longrightarrow$$

$$e = t[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n] \qquad e' = t[x_1 \mapsto e'_1, \ldots, x_n \mapsto e'_n]$$

$$\longrightarrow$$

$$\text{let f x} = \text{if} \ldots (e_1, \ldots, e_n) \ldots (e'_1, \ldots, e'_n) \ldots$$
$$\text{in scope}[f(a) \mapsto \text{let } x_1 \ldots x_n = f(a) \text{ in } t]$$

$\Longrightarrow$ No more square root or division in the function body

Function transformation

- Transformation order relies on a dependency graph build with variable, functions inputs and outputs:
  - f output depends on f input
  - f(...x...) $\Rightarrow$ f input depends on x
  - let f x = ...y... in ... $\Rightarrow$ f output depends on x
  - let x = ...f(e)... in ... $\Rightarrow$ x depends on f output

    $\vdots$

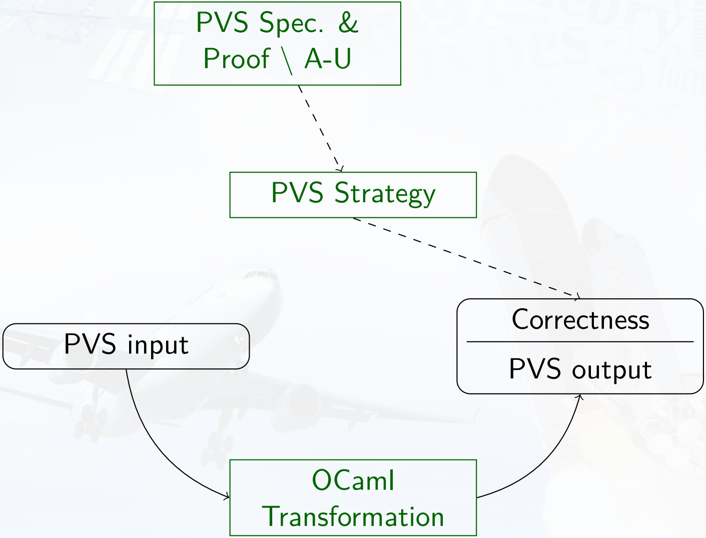
  when graph is acyclic we transform by following this graph

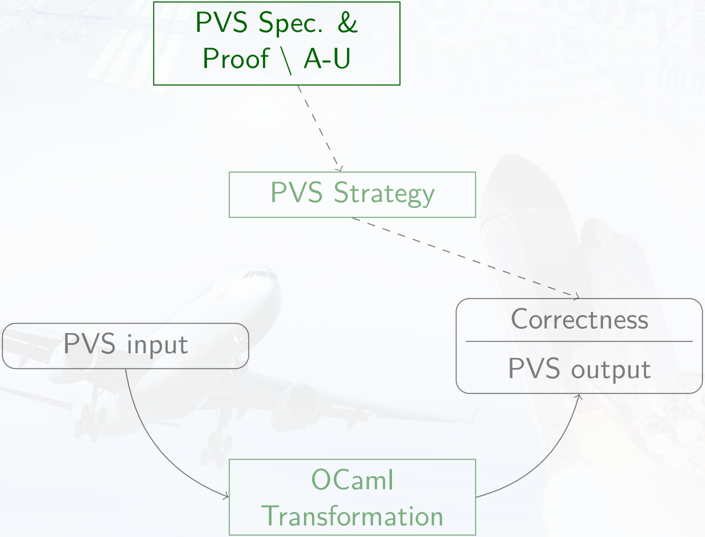
$/$ and $\sqrt{}$ elimination

Constrained Anti-unification

Certification with a proof assistant

- PVS Proof
- PVS Strategy
- Certifying


PVS Spec. & Proof \ A-U
PVS Strategy
Correctness
PVS output
PVS input
OCaml Transformation


PVS Spec. &
Proof \ A-U
PVS Strategy
PVS input
Correctness
PVS output
OCaml
Transformation

The PVS Proof Assistant

- Subtyping

  f (x : A | P(x)) : {y : B | Q(x,y)} = ...
  ... Case1 x = C(a) : if F then r1 else r2 ...

- Type Checking Conditions:

  f_TCC1 : ... $x = C(z) \Rightarrow F \Rightarrow Q(x,r1)$
  f_TCC2 : ... $x = C(z) \Rightarrow \neg F \Rightarrow Q(x,r2)$

PVS Specification

- Programs represented with an abstract datatype program (Inductive)

  program : DATATYPE = ...

  variable(va : string) : variable?

  bop(op: binop; pl : program, pr : program) : bop?

  $\vdots$

- Semantics of a program given by a sem function :

  sem(p : program, e : env) : value

  where value :=

  num(re : real) | boolv(bo : bool) | pair(value , value) | fail

- Transformation defined with the elim function :

  elim(p) : (pp : program | $\forall$ en,

  (nofail?(sem(p,en))) $\Rightarrow$ sem(p,en) = sem(pp,en))

PVS proof

- Soundness, preserves semantics:

  for every rule $r(p)$:

$$\forall Env : [\![\ p \ ]\!]_{Env} \neq Fail \Longrightarrow [\![\ p \ ]\!]_{Env} = [\![\ r(p) \ ]\!]_{Env}$$

PVS proof

- Soundness, preserves semantics:

  for every rule $r(p)$:

$$\forall Env : [\![\; p \;]\!]_{Env} \neq Fail \Longrightarrow [\![\; p \;]\!]_{Env} = [\![\; r(p) \;]\!]_{Env}$$

- Completeness, no square root in output:

  rules target different subtypes depending where $\sqrt{\ }$ and $/$ are allowed

PVS proof

- Soundness, preserves semantics:

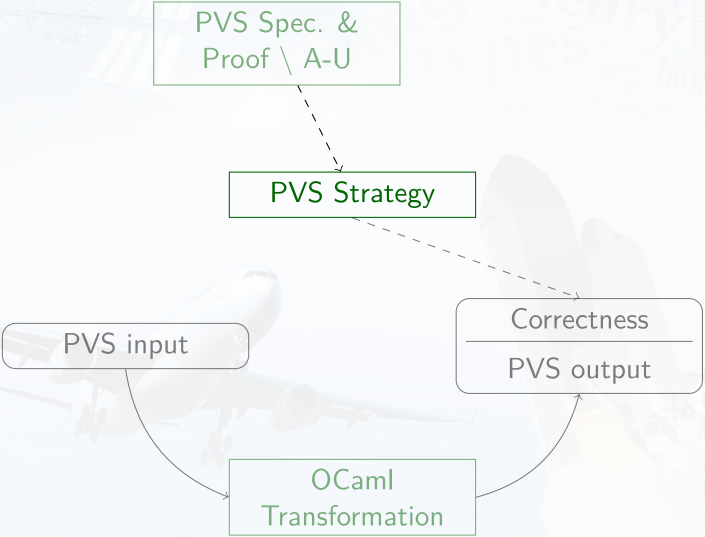
  for every rule $r(p)$:

  $$\forall Env : [\![\ p\ ]\!]_{Env} \neq Fail \Longrightarrow [\![\ p\ ]\!]_{Env} = [\![\ r(p)\ ]\!]_{Env}$$

- Completeness, no square root in output:

  rules target different subtypes depending where $\sqrt{}$ and $/$ are allowed

- Termination:
  - $\mathbb{B}$ expressions : number of square roots
  - programs : abstract datatype order


PVS Spec. &
Proof \ A-U
PVS Strategy
PVS input
Correctness
PVS output
OCaml
Transformation

$\sqrt{}$, $/$ and Theorem Provers

- SMT solvers or PVS strategies do not handle $\sqrt{}$ and $/$
- Use the PVS proof to define a PVS strategy

Formula transformation

```
  |----
{1} sqrt(a) > b
```

```
  |----
{1} -b > 0
{2} a - b * b > 0
```

Formula transformation

```
|----
{1} sqrt(a) > b
```

$\updownarrow$ grind

sem(gt(sqrt(A),B),env)

```
|----
{1} -b > 0
{2} a - b * b > 0
```

with env : [V -> value] such that env(A) = a and env(B) = b

Formula transformation

```
  |----
{1} sqrt(a) > b
```

grind

sem(gt(sqrt(A),B),env)

Correctness LEMMA

sem(elim(gt(sqrt(A),B)),env)

```
  |----
{1} -b > 0
{2} a - b * b > 0
```

with env : [V -> value] such that env(A) = a and env(B) = b

Formula transformation

```
|----
{1} sqrt(a) > b
```

```
|----
{1} -b > 0
{2} a - b * b > 0
```

grind

sem(gt(sqrt(A),B),env)

sem(or(gt(0,B),gt(A,times(B,B))),env)

Correctness LEMMA

eval-expr

sem(elim(gt(sqrt(A),B)),env)

with env : [V -> value] such that env(A) = a and env(B) = b

Formula transformation

```
|----
{1} sqrt(a) > b
```

grind

sem(gt(sqrt(A),B),env)

```
|----
{1} -b > 0
{2} a - b * b > 0
```

grind

sem(or(gt(0,B),gt(A,times(B,B))),env)

Correctness LEMMA

eval-expr

sem(elim(gt(sqrt(A),B)),env)

with env : [V -> value] such that env(A) = a and env(B) = b

Formula transformation

```
|----
{1} sqrt(a) > b
```

strategy

```
|----
{1} -b > 0
{2} a - b * b > 0
```

grind

grind

sem(gt(sqrt(A),B),env)

sem(or(gt(0,B),gt(A,times(B,B))),env)

Correctness LEMMA

eval-expr

sem(elim(gt(sqrt(A),B)),env)

with env : [V -> value] such that env(A) = a and env(B) = b

PVS strategy

F_p := gt(sqrt(A),B)
en(x) := if x = A then a
elsif x = B then b
else 0

lisp

```
|----
{1} sqrt(a) > b
```

```
{-1} sem(F_p,en)
  |----
{1} sqrt(a) > b
```

$\top$

case "sem(F_p,en)"

```
|----
{1} sem(F_p,en)
```

typepred "elim(F_p)"

```
|----
{1} sem(elim(F_p),en)
```

```
|----
{1} - b > 0
{2} a > b*b
```

Examples: (elim-sqrt)

```
  |----
{1} sqrt(x) > y

Rule?
```

Examples: (elim-sqrt)

```
  |----
{1} sqrt(x) > y

Rule? (elim-sqrt)
```

Examples: (elim-sqrt)

```
  |----
{1} sqrt(x) > y

Rule? (elim-sqrt)
```

$\vdots$

Examples: (elim-sqrt)

```
  |----
{1} sqrt(x) > y

Rule? (elim-sqrt)
```

⋮

```
  |----
{1} -y > 0
{2} -(y * y) + x > 0

Rule?
```

Examples: (elim-sqrt)

```
{-1} bb * bb - 4 * aa * cc >= 0
{-2} (-bb + sqrt(bb * bb - 4 * aa * cc)) / 2 >= 0
 |----
{1} cc > bb

Rule?
```

Examples: (elim-sqrt)

```
{-1} bb * bb - 4 * aa * cc >= 0
{-2} (-bb + sqrt(bb * bb - 4 * aa * cc)) / 2 >= 0
 |----
{1} cc > bb

Rule? (elim-sqrt)
```

Examples: (elim-sqrt)

```
{-1} bb * bb - 4 * aa * cc >= 0
{-2} (-bb + sqrt(bb * bb - 4 * aa * cc)) / 2 >= 0
 |----
{1} cc > bb

Rule? (elim-sqrt)
```

⋮

Examples: (elim-sqrt)

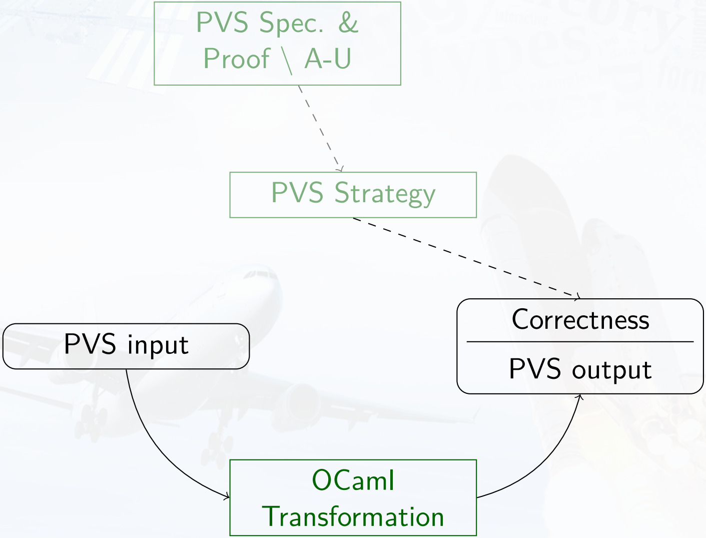
```
{-1} bb * bb - 4 * aa * cc >= 0
{-2} (-bb + sqrt(bb * bb - 4 * aa * cc)) / 2 >= 0
 |----
{1} cc > bb

Rule? (elim-sqrt)
```

⋮

```
{-1} -(4 * (-bb * -bb)) + 4 * -(4 * (aa * cc)) + 4 *
(bb * bb) >= 0 OR 2 * -bb >= 0
{-2} bb * bb - 4 * (aa * cc) >= 0
 |----
{1} cc > bb

Rule?
```


PVS Spec. &
Proof \ A-U
PVS Strategy
PVS input
Correctness
PVS output
OCaml
Transformation

Anti-Unification

- Termination problem of expression reduction:

$$\frac{\sum_{i=1}^{n} a_i \prod_{j_i=1}^{m_i} \sqrt{b_{j_i}}}{\sum_{i=1}^{n} c_i \prod_{j_i=1}^{m_i} \sqrt{d_{j_i}}}$$

$\Longrightarrow$ A-U algorithm is not certified

Anti-Unification

- Termination problem of expression reduction:

$$\frac{\sum\limits_{i=1}^{n} a_i \prod\limits_{j_i=1}^{m_i} \sqrt{b_{j_i}}}{\sum\limits_{i=1}^{n} c_i \prod\limits_{j_i=1}^{m_i} \sqrt{d_{j_i}}}$$

$\Longrightarrow$ A-U algorithm is not certified

- Certifying the result is quite easy:

Need to verify: $T\sigma_i = t_i$

A certifying Transformation

```
f(x1,y1 : posreal) : real = x1/y1

g(t : bool ,x,y : posreal) : real =
    IF t THEN f(x,(y + 1)) ELSE sqrt(x) + y ENDIF
```

is transformed into

```
f_e(x1, y1 : real) : {f_n, f_d : real |
   f_n / f_d = f((x1, y1))} = (x1, y1)

g_e(t : bool, x, y : real) :
 {g_1, g_2, g_d, sq_0 : real |
  (g_1 + g_2 * sqrt(sq_0)) / g_d = g((t, x, y))} =
        IF t
        THEN
         LET (f_n, f_d) = f_e((x, y + 1))
         IN (f_n, 0, f_d, 0)
        ELSE (y, 1, 1, x)
        ENDIF
```

Functions w. Sub-typing (Input)

letf f x : $A \to B$ =

    body;

... f(e) ... f(e')

$$\longrightarrow$$

$e = T_i[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ $\qquad$ $e' = T_i[x_1 \mapsto e'_1, \ldots, x_n \mapsto e'_n]$

$$\longrightarrow$$

letf f_e x_1 ... x_n : $A' \to \{\mathbf{y} : \mathbf{B} \mid \mathbf{y} = f(\mathbf{T_i})\} =$

    body[$x \mapsto T_i$];

... $f(e_1, \ldots, e_n)$ ... $f(e'_1, \ldots, e'_n)$

Functions w. Sub-typing (Output)

letf f_e x : $A \rightarrow \{y : B \mid y = \mathrm{f}(T_i)\} =$

if ... e ... e'...;

scope

$\longrightarrow$

e = T_o[$\mathrm{y}_1 \mapsto \mathrm{e}_1$,...,$\mathrm{y}_n \mapsto \mathrm{e}_n$] e' = T_o[$\mathrm{y}_1 \mapsto \mathrm{e'}_1$,...,$\mathrm{y}_n \mapsto \mathrm{e'}_n$]

$\longrightarrow$

letf f_e x : $A \rightarrow \{\mathbf{var}(\sigma_1) : \mathbf{B}' \mid \mathbf{T_o} = \mathrm{f}(\mathbf{T_i})\} =$

if ... (e_1,...,e_n) ... ($\mathrm{e'}_1$,...,$\mathrm{e'}_n$)...;

in scope[f(a) $\mapsto$ let y_1 ... y_n = f(a) in t]

Proof of the TCC

- PVS decomposes the function bodies in TCC generation

Proof of the TCC

- PVS decomposes the function bodies in TCC generation

  $\Longrightarrow$ Equalities of arithmetic expressions

Proof of the TCC

- PVS decomposes the function bodies in TCC generation

  $\Longrightarrow$ Equalities of arithmetic expressions

```
g_e(t : bool, x, y : real) :
 {g_1, g_2, g_d, sq_0 : real |
  (g_1 + g_2 * sqrt(sq_0)) / g_d = g((t, x, y))} =
        IF t
        THEN
         LET (f_n, f_d) = f_e((x, y + 1))
         IN (f_n, 0, f_d, 0)
        ELSE (y, 1, 1, x)
        ENDIF

g_e_TCC3: OBLIGATION
 FORALL (t: bool, x, y: real):
  NOT t IMPLIES (y + 1 * sqrt(x)) / 1 = g(t, x, y)
```

Proof of the TCC

- PVS decomposes the function bodies in TCC generation

  $\Longrightarrow$ Equalities of arithmetic expressions

```
g_e(t : bool, x, y : real) :
 {g_1, g_2, g_d, sq_0 : real |
  (g_1 + g_2 * sqrt(sq_0)) / g_d = g((t, x, y))} =
        IF t
        THEN
         LET (f_n, f_d) = f_e((x, y + 1))
         IN (f_n, 0, f_d, 0)
        ELSE (y, 1, 1, x)
        ENDIF

g_e_TCC3: OBLIGATION
 FORALL (t: bool, x, y: real):
  NOT t IMPLIES (y + 1 * sqrt(x)) / 1 = g(t, x, y)
```

- (elim-sqrt) and (grind-real) strategies terminate

A Certifying transformation

- `pvsio`: PVS parser + generation of code for OCaml

A Certifying transformation

- `pvsio`: PVS parser + generation of code for OCaml
- OCaml Implementation

A Certifying transformation

- `pvsio`: PVS parser + generation of code for OCaml
- OCaml Implementation
- Subtyping predicates

A Certifying transformation

- `pvsio`: PVS parser + generation of code for OCaml
- OCaml Implementation
- Subtyping predicates
- PVS Type checking conditions

A Certifying transformation

- `pvsio`: PVS parser + generation of code for OCaml
- OCaml Implementation
- Subtyping predicates
- PVS Type checking conditions
- PVS strategy (`elim-sqrt`)

A Certifying transformation

- pvsio: PVS parser + generation of code for OCaml
- OCaml Implementation
- Subtyping predicates
- PVS Type checking conditions
- PVS strategy (elim-sqrt)

$\Longrightarrow$ A Certifying transformation

Examples

Examples from the ACCoRD system (NASA):

- `cd2d.pvs` algorithm
  - Size: 2.9 kB $\longrightarrow$ 8kB
  - Memory: 17kB
- `trackline.pvs` algorithm
  - Size: 2.3 kB $\longrightarrow$ 13kB
  - Memory: 57 kB
- SMT example (Yices)
  - Size : 2.9 kB $\longrightarrow$ 12kB

Refinement by Transformation

$\mathbb{R}$-abstraction

$$\begin{tikzcd} \text{Specification} & P_{\mathbb{R}} \arrow[l, "\text{proof}"'] \arrow[d, dashed, leftrightarrow, "\underset{\simeq}{\textit{Abstraction}}"'] \\ & P \end{tikzcd}$$

Concrete world

Refinement by Transformation

$\mathbb{R}$-abstraction

$$\begin{tikzcd} \text{Specification} & P_{\mathbb{R}} \arrow[l, "\text{proof}"'] \arrow[d, dashed, leftrightarrow, "\textit{Abstraction}\ \simeq"'] & \\ & P \arrow[r, "\text{Transformation}"'] & P' \end{tikzcd}$$

Concrete world

Refinement by Transformation

$\mathbb{R}$-abstraction

$$\begin{tikzcd} \text{Specification} & P_{\mathbb{R}} \arrow[l, "\text{proof}"'] \arrow[d, dashed, leftrightarrow, "\text{Abstraction}\ \simeq"'] & & P'_{\mathbb{R}} \arrow[d, dashed, leftrightarrow, "\text{Abstraction}\ ="] \\ & P \arrow[rr, "\text{Transformation}"'] & & P' \end{tikzcd}$$

Concrete world

Refinement by Transformation

$\mathbb{R}$-abstraction

$$
\begin{tikzcd}
\text{Specification} & P_{\mathbb{R}} \arrow[l, "\text{proof}"'] \arrow[r, leftrightarrow, "\overset{\text{Semantics equality}}{=}"] \arrow[d, dashed, leftrightarrow, "\underset{\simeq}{\textit{Abstraction}}"'] & P'_{\mathbb{R}} \arrow[d, dashed, leftrightarrow, "\underset{=}{\textit{Abstraction}}"] \\
& P \arrow[r, "\text{Transformation}"'] & P'
\end{tikzcd}
$$

Concrete world

Implementations

- OCamL implementation:
  - Efficient anti-unification
  - Function transformation
  - Subtyping predicates
- PVS specification:
  - Proves Correctness semantics and elimination
  - Incomplete (anti-unification axiomatized)
- PVS strategy:
  - By computational reflection
  - Allows the use of decision procedures

Future work

- Complete PVS proof (AU + Functions)
  - A certified transformation
- Performance
- Extend language: cyclic dependency graph, bounded loops
  - Anti-unification fixpoint
- Other operations

Conclusion

Program transformation:

- delay computation of inexact operations
- protect the control flow
- only manipulate terms that can be exactly computed

Conclusion

Program transformation:

- delay computation of inexact operations
- protect the control flow
- only manipulate terms that can be exactly computed

$\Longrightarrow$ Continuous Programs

Questions ?