

Translating PVS to C

SRI International

Gaspard Férey

Ecole Polytechnique

September 1st, 2014

Table of Contents

- 1 Context
- 2 Static analysis
 - The update issue
 - Reference tracking
 - Mutable analysis
- 3 The PVS Translator
 - Architecture
 - Implementation
- 4 Conclusion
 - Demonstration

Table of Contents

- 1 Context
- 2 Static analysis
 - The update issue
 - Reference tracking
 - Mutable analysis
- 3 The PVS Translator
 - Architecture
 - Implementation
- 4 Conclusion
 - Demonstration

What is PVS ?

Prototype Verification System

- A specification language
 - ▶ Expressive syntax
 - ▶ Rich type system
 - ★ uninterpreted types
 - ★ abstract data types
 - ★ predicate subtypes
 - ★ etc
 - ▶ Parameterized theories
- A semi automated theorem prover
 - ▶ Higher order logic
 - ▶ Type system, type-correctness conditions, judgments, ...
 - ▶ Theorems, properties, ...
 - ▶ SMT solvers and other tools integrated
- A functional programming language (?)

Why translate PVS ?

- To be able to execute PVS
 - ▶ Testing
 - ▶ Debugging
- To integrate high-insurance PVS code into systems

Table of Contents

- 1 Context
- 2 Static analysis
 - The update issue
 - Reference tracking
 - Mutable analysis
- 3 The PVS Translator
 - Architecture
 - Implementation
- 4 Conclusion
 - Demonstration

Update expression

- In functional programming languages,

$$E := A \text{ WITH } [(x) := y]$$

$$\text{refers } i \mapsto \begin{cases} A(i) & \text{if } i \neq x \\ y & \text{if } i = x \end{cases}$$

- In imperative languages
 - ▶ $A[x := y]$ a non destructive update using a copy of A .
 - ▶ $A[x \leftarrow y]$ a destructive, in-place update of the aggregate structure representing A .

Two dangers

- Unsafe occurrence

`LET B = A WITH[(0) := 0] IN B(0) + A(0)`

The array represented by A is used later in the code.

- Trapped reference

`LET A = B(0) IN f(A WITH [(0) := 0], B(0))`

B is affected by a destructive update of the array represented by A.

Previous analysis

Shankar's analysis relies on sets of *variables*

- Av active variables
- Ov output variables
- Fv free variables
- Lv live variables in an *update context*

Cerny and Shankar's analysis adds *flow analysis*.

The intermediate language

Syntax

- Integers, nil pointer
- Variables (X, x, y, \dots)
- `newArray(x, y)`
- $X[x := y]$
- $X[x \leftarrow y]$
- $X[x]$
- `let $x = a$ in e`
- `if(x) e_1 else e_2`
- `f(x_1, \dots, x_n)`

Memory state representation:

- Variables space Var
- Reference space \mathcal{R}
- Value space $\mathcal{V} := \mathbb{N} \cup \mathcal{R}$
- Store $R : \mathcal{R} \rightarrow \mathcal{V}$
- Stack $S : Var \rightarrow \mathcal{V}$

Evaluation context

- hole $\{\}$
- `let $x = \{\}$ in e`
- $E_1\{E_2\}$

The intermediate language

Operational semantics

Simple reduction rules:

$$\begin{aligned} \langle x | R, S \rangle &\rightarrow \langle S(x) | R, S \rangle \\ \langle x[y] | R, S \rangle &\rightarrow \langle R(S(x))(S(y)) | R, S \rangle \\ \langle \text{if}(x) \ a \ \text{else} \ b | R, S \rangle &\rightarrow \begin{cases} \langle b | R, S \rangle & \text{if } S(x) = 0 \\ \langle a | R, S \rangle & \text{otherwise} \end{cases} \end{aligned}$$

Introducing variables

$$\begin{aligned} \langle f(x_1, \dots, x_n) | R, S \rangle &\rightarrow \langle \text{pop}([f]) | R, \left(\biguplus_{1 \leq i \leq n} f_i \mapsto S(x_i) \right) :: S \rangle \\ \langle \text{let } x = v \text{ in } e | R, S \rangle &\rightarrow \langle \text{pop}(e) | R, (x \mapsto v) :: S \rangle \\ \langle \text{pop}(v) | R, S \rangle &\rightarrow \langle v | R, \text{pop}(S) \rangle \end{aligned}$$

The intermediate language

Operational semantics

Modifying the store

$$\langle \text{newArray}(x, y) \mid R, S \rangle \rightarrow \langle r \mid R \uplus (r \mapsto (S(y))_{0 \leq i < S(x)}) , S \rangle$$

where r is a fresh pointer

$$\langle X[x := y] \mid R, S \rangle \rightarrow \langle r \mid R', S \rangle$$

where r fresh pointer

$$\text{and } R' := R \uplus (r \mapsto A)$$

$$\text{and } A := R(S(X)) \uplus (S(x) \mapsto S(y))$$

$$\langle X[x \leftarrow y] \mid R, S \rangle \rightarrow \langle X \mid R', S \rangle$$

where $R' := R \uplus (S(X) \mapsto A)$

$$\text{and } A := R(S(X)) \uplus (s(x) \mapsto S(y))$$

Reference graph

For a context in the body of a function

```
f(A, B) =  
  let C = if(A[0] = 1) then A else B in  
  let D =  
    let E = C in E[0] := 0 in  
  D[0] + A(0)
```

we can define the reference graph $\mathcal{G}(R)(r)$ as

$$\begin{aligned} r &\in \mathcal{G}(r) \\ \mathcal{R} \cap R(\mathcal{G}(r)) &\subset \mathcal{G}(r) \end{aligned}$$

We are interested in

$$\bigcup_R S^{-1}(\mathcal{G}(R)(S(x)))$$

Reference graph

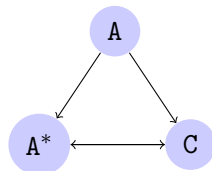
Definition of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
          else X in  
  let G = F[0 := 0] in  
  let H = if B[0] then G  
          else B in  
  H[0 := B[0]]
```

Reference graph

```
f(A, B) =  
  let C = A[0]      in  
  let D = { C }     in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
              else X in  
  let G = F[0 := 0] in  
  let H = if B[0] then G  
              else B in  
  H[0 := B[0]]
```

Reference graph



Variables lives in the context

$\{B, D, E, F, G, H, X\}$

Destructive update would impose too many requirements to f .

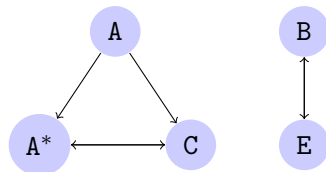
Reference graph

Definition of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = { E }     in  
  let F = if X[0] then D  
           else X in  
  let G = F[0 := 0] in  
  let H = if B[0] then G  
           else B in  
  H[0 := B[0]]
```

B is live in the context.

Reference graph



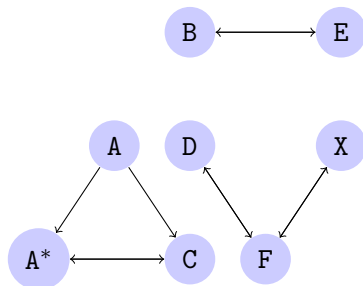
Variables lives in the context

$\{B, C, D, F, G, H, X\}$

Reference graph

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D else X in  
  
  let G = { F }      in  
  let H = if B[0] then G else B in  
  
  H[0 := B[0]]
```

Reference graph



Variables lives in the context

$\{B, C, G, H\}$

Destructive update possible.

Reference graph

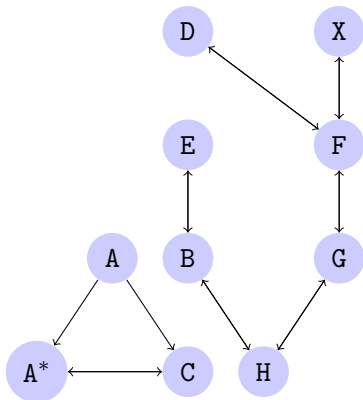
```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
           else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
           else B in  
  { H }
```

Variables lives in the context

{ }

Destructive update possible.

Reference graph



The flags

We implement an approximation of the analysis using three flags

- **mutable**

- ▶ Variable: Every other variable that may point to that variable is not live.
- ▶ Argument: Variables passed as this argument must be flagged **mutable** and **safe** .
- ▶ Function: The result of a call to that function is "fresh" .

- **dupl**

- ▶ Argument: Possible active variable in a call to this function.
- ▶ Expression: May be aliased to the return value.

- **safe**

- ▶ Last of occurrence of a variable

The rules

- Two versions for each function
 - ▶ destructive
 - ▶ non destructive
- **mutable** flag:
 - ▶
- **dupl** flag:
 - ▶

Example

Destructive version of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
           else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
           else B in  
  H[0 <- 0]
```

Example

C no flags

Destructive version of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
          else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
          else B in  
  H[0 <- 0]
```

Example

Destructive version of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
          else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
          else B in  
  H[0 <- 0]
```

C no flags

D **mutable**

⇐ non destructive update

Example

Destructive version of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
          else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
          else B in  
  H[0 <- 0]
```

C no flags

D **mutable**

⇐ non destructive update

E no flags

Example

Destructive version of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
           else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
           else B in  
  H[0 <- 0]
```

C no flags

D **mutable**

⇐ non destructive update

E no flags

X **mutable**

⇐ non destructive update

Example

Destructive version of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
           else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
           else B in  
  H[0 <- 0]
```

C no flags

D **mutable**

⇐ non destructive update

E no flags

X **mutable**

⇐ non destructive update

F **mutable**

⇐ D and X **safe** and **mutable**

Example

Destructive version of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
           else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
           else B in  
  H[0 <- 0]
```

C no flags

D **mutable**

⇐ non destructive update

E no flags

X **mutable**

⇐ non destructive update

F **mutable**

⇐ D and X **safe** and **mutable**

G **mutable**

⇐ F **safe** and **mutable**

Example

Destructive version of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
          else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
          else B in  
  H[0 <- 0]
```

C no flags

D **mutable**

⇐ non destructive update

E no flags

X **mutable**

⇐ non destructive update

F **mutable**

⇐ D and X **safe** and **mutable**

G **mutable**

⇐ F **safe** and **mutable**

H **mutable**

⇐ G and B **safe** and **mutable**

Example

Destructive version of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
           else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
           else B in  
  H[0 <- 0]
```

C no flags

D **mutable**

⇐ non destructive update

E no flags

X **mutable**

⇐ non destructive update

F **mutable**

⇐ D and X **safe** and **mutable**

G **mutable**

⇐ F **safe** and **mutable**

H **mutable**

⇐ G and B **safe** and **mutable**

⇒ Arguments B gets **mutable**

Example

Destructive version of f

```
f(A, B) =  
  let C = A[0]      in  
  let D = C[0 := 0] in  
  let E = g(B)      in  
  let X = E[0 := 0] in  
  let F = if X[0] then D  
          else X in  
  let G = F[0 <- 0] in  
  let H = if B[0] then G  
          else B in  
  H[0 <- 0]
```

C no flags

D **mutable**

⇐ non destructive update

E no flags

X **mutable**

⇐ non destructive update

F **mutable**

⇐ D and X **safe** and **mutable**

G **mutable**

⇐ F **safe** and **mutable**

H **mutable**

⇐ G and B **safe** and **mutable**

⇒ Arguments B gets **mutable**

H **safe**

A reference counting GC

We complete the static analysis with a reference counting garbage collector (GC)

- Easy to implement in C (hashtable of pointers)
- Memory freed soon
- The reference count can allow safe update to be made destructive

Table of Contents

- 1 Context
- 2 Static analysis
 - The update issue
 - Reference tracking
 - Mutable analysis
- 3 The PVS Translator
 - Architecture
 - Implementation
- 4 Conclusion
 - Demonstration

Translation steps

- Typechecking: PVS typechecker
 - ▶ TCCs are generated
- Lexical and syntactic analysis: PVS lexer and parser
 - ▶ PVS \longrightarrow CLOS representation
- Translation:
 - ▶ CLOS representation \longrightarrow intermediate language

Translation steps (2)

- Static analysis:
 - ▶ destructive updates added
- Optimizations: Several passes
 - ▶ Choosing C types
 - ▶ Declaring and freeing variables
- Code generation:
 - ▶ intermediate language \longrightarrow compilable C code

Implementation details

- In Common Lisp
- Directly integrated to PVS (soon)
- Require the GMP library to run C code

Table of Contents

- 1 Context
- 2 Static analysis
 - The update issue
 - Reference tracking
 - Mutable analysis
- 3 The PVS Translator
 - Architecture
 - Implementation
- 4 Conclusion
 - Demonstration

- Working compiler
 - ▶ Output C code compilable with gcc, CompCert (?)
 - ▶ Use GMP library to represent PVS big integers
- Promising analysis
- Optimization efficient (for simple programs)
 - ▶ 100 times faster than Ground Evaluator

Left to be done

- More work on static analysis
 - ▶ Reference counting
 - ▶ Publication
- Less approximate implementation
- Debugging, optimizing Lisp code
- Wider subset of PVS
 - ▶ Closures
 - ▶ More types
 - ▶ ...

Demonstration

The program

- We compute the array T

$$T(0) = 9876$$

$$T(i) = (12345 * T(i)) \bmod 59557 \quad \text{for } 0 < i < 10.000$$

- We sort T it using an insertion sort.

```
insert(A, i, v) =  
  if    i = 0 or v >= A[i-1]  
  then  A[i := v]  
  else  insert( A[i := A[i-1]], i-1, v)
```

```
sort(A, n) =  
  if    n = 100000  
  then  A  
  else  sort( insert(A, A[n], n), n+1 )
```

```
min(A) = sort(A, 0)[0]
```

- Complexity: time $O(n^2)$ space $O(n)$

Questions ?

