Translating PVS to C

SRI International

Gaspard Férey

Ecole Polytechnique

September 1st, 2014

Table of Contents

Table of Contents

What is PVS ?

- A specification language
  - Typed expression definition
  - Theories, datatypes, ...
- A semi automated theorem prover
  - Higher order logic
  - Type system, judgments, ...
  - Theorems, properties, ...
  - SMT solvers integrated, tools, ...
- A functional programming language (?)

Why translate PVS ?

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

Table of Contents

Update expression

- In functional programming languages,

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

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 <- 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`, ...)
- `newArray(x, y)`
- `X[(x) := y]`
- `X[(x) <- y]`
- `X[x]`
- `let x =` a `in` e
- `if(x)` e_1 `else` e_2
- `f(`x_1`, ... ,`x_n`)`

Memory state representation:

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

Evaluation context

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

Simple reduction rules:

$$
\begin{array}{rcl}
< x|R, S > & \rightarrow & < S(x)|R, S > \\
< x\,[y]\,|R, S > & \rightarrow & < R(S(x))(S(y))|R, S > \\
< \texttt{if(x)}\ a\ \texttt{else}\ b|R, S > & \rightarrow & \begin{cases} < b|R, S > & \text{if } S(x) = 0 \\ < a|R, S > & \text{otherwise} \end{cases}
\end{array}
$$

Introducing variables

$$
\begin{array}{rcl}
< f\,(x_1, \ldots, x_n)\,|R, S > & \rightarrow & < \texttt{pop}([f])\,|R, \left(\biguplus_{1 \leq i \leq n} f_i \mapsto s(x_i) \right) :: S > \\
< \texttt{let}x\ \texttt{=}\ v\ \texttt{in}\ e|R, S > & \rightarrow & < \texttt{pop}(e)\,|R, (x \mapsto v) :: S > \\
< \texttt{pop}(v)\,|R, S > & \rightarrow & < v|R, pop(S) >
\end{array}
$$

Modifying the store

$$
\begin{array}{rcl}
<\texttt{newArray}(x,y)|R,S> & \rightarrow & <r|R \uplus \left(r \mapsto (0)_{0\leq i<S(x)}\right), S> \\
 & \text{where} & r \text{ is a fresh pointer} \\
<X\texttt{[}(x)\ \texttt{:=}\ y\texttt{]}|R,S> & \rightarrow & <r|R',S> \\
 & \text{where} & r \text{ fresh pointer} \\
 & \text{and} & R' := R \uplus (r \mapsto A) \\
 & \text{and} & A := R(S(X)) \uplus (S(x) \mapsto S(y)) \\
<X\texttt{[}(x)\ \texttt{<-}\ y\texttt{]}|R,S> & \rightarrow & <X|R',S> \\
 & \text{where} & R' := R \uplus (S(X) \mapsto A) \\
 & \text{and} & A := R(S(X)) \uplus (s(x) \mapsto S(y))
\end{array}
$$

Reference graph

In the body of a function [example] we can define the reference graph [definition]

Reference graph

We can keep track of it [two columns example column 1: PVS code column 2: Tree + live variable]
The condition to destructively update is empty intersection

The flags

We implement an approximation of the analysis using 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 output variable of a call to this function
  - Expression: May be aliased to the return value.
- **safe**
  - Last of occurrence of a variable

The rules

mutable flag:

-

dupl flag:

-

A reference counting GC

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

Table of Contents

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

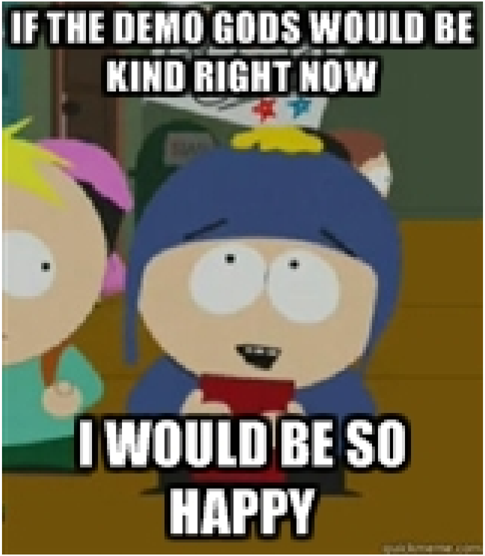
Example

Table of Contents

Conclusion

Questions ?

Demonstration



Questions ?