Building Verification Condition Generators by Compositional Extension

A. J. van Leeuwen

Department of Information and Computing Sciences
Universiteit Utrecht

2005 11 29

Universiteit Utrecht

Why Build Verification Condition Generators Compositionally?

We want mechanized tool support for reasoning over languages, but:

- Languages change
- Logics for languages must change along
- Mechanized tools for these logics must change along
- Keeping trust in tools is hard
- Changing such tools diminishes trust
- Compositionality improves clarity, ease of correct modification and therefore trust

Oh, and it's fun too!

Consider language L:

$$
\begin{aligned}
\mathit{Stmt} \rightarrow\ & \mathit{Variable} := \mathit{Expr} \\
\mid\ & \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \{\,\mathit{Stmt}\,\}\ \textbf{else}\ \{\,\mathit{Stmt}\,\} \\
\mid\ & \textbf{inv}\ \mathit{Expr}\ \textbf{while}\ \mathit{Expr}\ \textbf{do}\ \{\,\mathit{Stmt}\,\} \\
\mid\ & \mathit{Stmt}; \mathit{Stmt}
\end{aligned}
$$

with these derivation rules for sufficient precondition predicate transformer semantics:

$$
\begin{aligned}
&\mathrm{pre}\ (x\ :=\ e)\ q\ =\ q[e/x] \\
&\mathrm{pre}\ (S_1;\ S_2)\ q\ =\ \mathrm{pre}\ (S_1)\ (\mathrm{pre}\ (S_2)\ q) \\
&\mathrm{pre}\ (\textbf{if}\ g\ \textbf{then}\ \{S_1\}\ \textbf{else}\ \{S_2\})\ q\ = \\
&\qquad\qquad\qquad\qquad \mathit{if}\ g\ \mathit{then}\ \mathrm{pre}\ (S_1)\ q\ \mathit{else}\ \mathrm{pre}\ (S_2)\ q
\end{aligned}
$$

$$
\frac{
\begin{array}{l}
p\ =\ \mathrm{pre}\ (S)\ i \\
i \wedge \neg g \Rightarrow q \\
i \wedge g \Rightarrow p
\end{array}
}{
\mathrm{pre}\ (\textbf{inv}\ i\ \textbf{while}\ g\ \textbf{do}\ \{S\})\ q\ =\ i
}
$$

Universiteit Utrecht

Function: $pvcg :: Stmt \rightarrow Expr \rightarrow Expr$

Function: $pvcg :: Stmt \rightarrow (Expr, [Expr]) \rightarrow (Expr, [Expr])$

Function: $pvcg :: Stmt \rightarrow m\ Expr \rightarrow m\ Expr$

Function: $pvcg :: Stmt \to Expr \to m\ Expr$

Decouple from recursion, abstract away gathering:

$$
\begin{aligned}
pvcg\_Assign\ x\ e &= \lambda q \to return\ (subst\ (x, e)\ q) \\
pvcg\_Seq\ p\_s1\ p\_s2 &= \lambda q \to p\_s2\ q \gg\!= \lambda p' \to \\
&\qquad p\_s1\ p' \gg\!= \lambda p \to \\
&\qquad return\ p \\
pvcg\_IfThenElse\ g\ s1\ s2 &= \quad \text{-- similar to others} \\
pvcg\_InvWhile\ i\ g\ p\_body &= \\
&\quad \lambda q \to p\_body\ i \gg\!= \lambda p \to \\
&\qquad record\ (i \wedge \neg\, g \to q) \gg\!= \lambda\_ \to \\
&\qquad record\ (i \wedge g \quad \to p) \gg\!= \lambda\_ \to \\
&\qquad return\ i
\end{aligned}
$$

$$
pvcg = foldStmt\ (pvcg\_Assign, pvcg\_Seq, pvcg\_IfThenElse, \\
\quad pvcg\_InvWhile)
$$

Universiteit Utrecht

Add exception handling constructs to the language:

$$
\begin{array}{lll}
\mathit{Stmt} \rightarrow & & \text{-- existing alternatives} \\
& \mid & \textbf{try}\ \ \mathit{Stmt}\ \ \textbf{catch}\ \ \mathit{Stmt} \\
& \mid & \textbf{raise}
\end{array}
$$

Add exception handling constructs to the language:

$$
\begin{array}{ll}
\mathit{Stmt} \rightarrow & \text{-- existing alternatives} \\
\quad\quad\mid & \textbf{try}\ \ \mathit{Stmt}\ \ \textbf{catch}\ \ \mathit{Stmt} \\
\quad\quad\mid & \textbf{raise}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{pvcg\_Raise} & = \lambda q \rightarrow & \mathit{getPostE} \\
\mathit{pvcg\_TryCatch}\ \mathit{p\_s1}\ \mathit{p\_s2} & = \lambda q \rightarrow & \mathit{getPostE} \gg\!= \quad \lambda q' \rightarrow \\
 & & \mathit{p\_s2}\ q \gg\!= \quad \lambda p' \rightarrow \\
 & & \mathit{setPostE}\ p' \gg\!= \lambda\_ \rightarrow \\
 & & \mathit{p\_s1}\ q \gg\!= \quad \lambda p \rightarrow \\
 & & \mathit{setPostE}\ q' \gg\!= \lambda\_ \rightarrow \\
 & & \mathit{return}\ p
\end{array}
$$

$$
\begin{array}{l}
\mathit{pvcg} = \mathit{foldStmt}\ (\mathit{pvcg\_Assign}, \mathit{pvcg\_Seq}, \mathit{pvcg\_IfThenElse}, \\
\quad \mathit{pvcg\_InvWhile}, \mathit{pvcg\_Raise}, \mathit{pvcg\_TryCatch})
\end{array}
$$

Choosing the monad

- The monad is an unspecified paramater
- Specifying it directly is not very compositional
- We can use monad transformers, distributive laws or *coproducts*

- Monad transformers fix the order of composition
- Distributive laws cannot always be given
- Coproducts are more general than both
- Coproducts require a third monad to encapsulate semantics
- Given mapping from each element to the third monad, the mapping from the coproduct can be derived

Summarizing

We have introduced an approach to implementing verification condition generators that:

- Can deal with changes to the language
- Can deal with changes to the underlying domain
- Without having to change things that have already been specified