

Building Verification Condition Generators by Compositional Extension

A. J. van Leeuwen

Department of Information and Computing Sciences
Universiteit Utrecht

2005 11 29



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 :

$$\begin{array}{l} Stmt \rightarrow Variable := Expr \\ \quad | \text{ if } Expr \text{ then } \{ Stmt \} \text{ else } \{ Stmt \} \\ \quad | \text{ inv } Expr \text{ while } Expr \text{ do } \{ Stmt \} \\ \quad | Stmt; Stmt \end{array}$$

with these derivation rules for sufficient precondition predicate transformer semantics:

$$\begin{array}{l} \text{pre } (x := e) q = q[e/x] \\ \text{pre } (S_1; S_2) q = \text{pre } (S_1) (\text{pre } (S_2) q) \\ \text{pre } (\text{if } g \text{ then } \{S_1\} \text{ else } \{S_2\}) q = \\ \qquad \qquad \qquad \text{if } g \text{ then } \text{pre } (S_1) q \text{ else } \text{pre } (S_2) q \end{array}$$
$$\frac{\begin{array}{l} p = \text{pre } (S) i \\ i \wedge \neg g \Rightarrow q \\ i \wedge g \Rightarrow p \end{array}}{\text{pre } (\text{inv } i \text{ while } g \text{ do } \{S\}) q = i}$$


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

Decouple from recursion, abstract away gathering:

$$pvcg_Assign\ x\ e = \lambda q \rightarrow return\ (subst\ (x, e)\ q)$$
$$pvcg_Seq\ p_s1\ p_s2 = \lambda q \rightarrow p_s2\ q \gg\gg \lambda p' \rightarrow \\ p_s1\ p' \gg\gg \lambda p \rightarrow \\ return\ p$$
$$pvcg_IfThenElse\ g\ s1\ s2 = \quad --\ \text{similar to others}$$
$$pvcg_InvWhile\ i\ g\ p_body = \\ \lambda q \rightarrow p_body\ i \gg\gg \lambda p \rightarrow \\ record\ (i \wedge \neg g \rightarrow q) \gg\gg \lambda_ \rightarrow \\ record\ (i \wedge g \rightarrow p) \gg\gg \lambda_ \rightarrow \\ return\ i$$
$$pvcg = foldStmt\ (pvcg_Assign, pvcg_Seq, pvcg_IfThenElse, \\ pvcg_InvWhile)$$


Add exception handling constructs to the language:

```
Stmt →  -- existing alternatives  
      | try Stmt catch Stmt  
      | raise
```



Choosing the monad

- ▶ The monad is an unspecified parameter
- ▶ 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

