Programming language semantics

Paul Jackson

School of Informatics
University of Edinburgh

Formal Verification
Spring 2018

Using maths to verify software

- First need ability to construct mathematical models of programs
  - Highly non-trivial – most programming languages are complex and have no formal description
  - Particularly difficult when handling concurrency
  - Most focus on functional behaviour. Only few handle performance
- To enable the automation of proof we then need systematic recipes for carrying out proof
- Notion of *proof* is broad: it might involve
  - Applying rules of a program calculus
  - Computing data-structures (e.g. BDDs in symbolic model checking)

Common themes in FV approaches

- A frequent theme is reducing program correctness to checking validity of formulas in propositional or first-order logic.
- This enables use of automated theorem prover technology
  - SAT solvers in bounded model checking
  - SMT solvers with the *weakest precondition* approach taken by the SPARK FV tools
- Just as with compilers, another theme is first translating to a simpler intermediate program language before engaging the core generation of logical formulas.
  - SPARK GNATprove tool uses front end of GNAT compiler.

IMP - a toy imperative programming language

- Numbers **N** $m, n ::= \ldots \mid -1 \mid 0 \mid 1 \mid 2 \mid \ \ldots$
- Variables **Var** x, y
- Arithmetic expressions **Aexp**

$$a ::= n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

- Boolean expressions **Bexp**

$$b ::= \textbf{true} \mid \textbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

- Commands **Com**

$$\begin{aligned} c ::= \quad & \textbf{skip} \mid x := a \mid c_o \ ; \ c_1 \\ & \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c \end{aligned}$$

This is abstract syntax, ignoring parentheses

Operational semantics

- Define a set of states Σ as all functions $\sigma : \mathbf{Var} \rightarrow \mathbf{N}$
- Use relations to define how
  - expressions evaluate to values in a given state
  - commands execute, changing the program state.

Evaluation of arithmetic expressions

Use 3 place relation

$$\langle a, \sigma \rangle \rightarrow n$$

where a is an arithmetic expression, σ the current state and n the value of the expression.

Relation defined in syntax-directed way:

$$\langle n, \sigma \rangle \rightarrow n$$

$$\langle x, \sigma \rangle \rightarrow \sigma(x)$$

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n} \text{ where } n \text{ is } n_0 + n_1$$

Similarly can define relation for Boolean expressions.

Big-step operational semantics for IMP

Relation

$$\langle c, \sigma \rangle \to \sigma'$$

expresses that command c executed in initial state σ terminates in final state σ'.

$$\langle \textbf{skip}, \sigma \rangle \to \sigma$$

$$\frac{\langle a, \sigma \rangle \to m}{\langle x := a, \sigma \rangle \to \sigma[m/x]}$$

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0 \,;\; c_1, \sigma \rangle \to \sigma'}$$

Big-step operational semantics cont.

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma'' \rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma'}$$

Program specifications

A basic way of specifying desired program behaviour is using preconditions and postconditions.

We commonly write

$$\{P\}\ c\ \{Q\}$$

to express that if program c is started in a state satisfying precondition P and if it terminates, it will terminate in a state satisfying postcondition Q.

$\{P\}\ c\ \{Q\}$ is known as a Hoare triple.

It can be defined semantically in terms of the big-step operational semantics relation

$$\models \{P\}c\{Q\} \ \doteq\ \textit{for all}\ \sigma, \sigma' \in \Sigma\ \ \textit{if}\ \sigma \models P\ \textit{and}\ \langle \sigma, c \rangle \to \sigma'\ \textit{then}\ \sigma' \models Q$$

Doing proofs directly with the execution relation $\to$ is tedious.

Hoare logics

An alternative to reasoning directly with the execution relation is using a calculus with Hoare triples.

An example rule:

$$\frac{\{P\}\, c_0\, \{R\} \quad \{R\}\, c_1\, \{Q\}}{\{P\}\, c_0 \,;\, c_1\, \{Q\}}$$

Such calculi are known as Hoare logics.

Hoare logics can be good for paper proofs and proofs using an interactive theorem prover, but are not the best for automation.

In the above rule, what is a recipe for R?

Weakest pre-condition based approaches are better.

Weakest pre-condition

The weakest pre-condition function $\mathrm{WP}(,)$ can be defined semantically:

$$\mathrm{WP}(c, Q) \doteq \{\sigma \mid \textit{for all } \sigma' \textit{ if } \langle c, \sigma\rangle \to \sigma' \textit{ then } \sigma' \models Q\}$$

where we identify predicates with the sets of states that satisfy them.

$\mathrm{WP}(,)$ is closely related to Hoare triples. We have

$$(\textit{for all } \sigma \textit{ if } \sigma \models P \textit{ then } \sigma \in \mathrm{WP}(c, Q)) \quad \textit{iff} \quad \models \{P\}\, c\, \{Q\}$$

and in particular

$$\{\mathrm{WP}(c, Q)\}\, c\, \{Q\}$$

$\mathrm{WP}(c, Q)$ is indeed the weakest pre-condition of c and Q.

How weakest pre-conditions can be used for verification

If we can compute $\mathrm{WP}(c, Q)$ as a formula, given formula for Q, then proving the predicate logic formula

$$\forall \bar{x}.\ P \Rightarrow \mathrm{WP}(c, Q)$$

is sufficient for establishing

$$\{P\}\ c\ \{Q\}$$

Here

- The $\forall \bar{x}$ is a quantification over all the variables in **Var** – the syntactic equivalent of quantifying over all states
- $\forall \bar{x}.\ P \Rightarrow \mathrm{WP}(c, Q)$ is called a verification condition or VC

Weakest precondition equations

$$\begin{aligned}
\mathrm{WP}(\textbf{skip}, Q) &= Q \\
\mathrm{WP}(x := a, Q) &= Q[x \mapsto a] \\
\mathrm{WP}(c_0\ ;\ c_1, Q) &= \mathrm{WP}(c_0, \mathrm{WP}(c_1, Q)) \\
\mathrm{WP}(\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, Q) &= (b \Rightarrow \mathrm{WP}(c_0, Q)) \\
&\quad \wedge (\neg b \Rightarrow \mathrm{WP}(c_1, Q)) \\
\mathrm{WP}(\textbf{while}\ b\ \textbf{do}\ c, Q) &= (b \Rightarrow \mathrm{WP}(c\ ;\ \textbf{while}\ b\ \textbf{do}\ c, Q)) \\
&\quad \wedge (\neg b \Rightarrow Q)
\end{aligned}$$

Here now the left and right hand sides of the equations are Boolean expressions in the program variables.

Given formula Q and c without while loops, equations specify how to compute $\mathrm{WP}(c, Q)$ as a formula.

If c has while loops, computation would not terminate.

Addressing the loop issue

Rough idea:

1. Add a loop invariant assertion to every loop of a program c
   - These assertions cut the control flow of c into loop-free segments

2. Show $\{P\}\, c\, \{Q\}$ by showing $\{P'\}\, c'\, \{Q'\}$ for each segment c' making up c.
   - Each P' is either P or a loop invariant.
   - Each Q' is either a loop invariant or Q.

3. Show $\{P'\}\, c'\, \{Q'\}$ by proving

$$\forall \bar{x}.\ P' \Rightarrow \mathrm{WP}(c', Q')$$

A detail:

Segments might have multiple initial and final points.
Must check $\{P'\}\, c''\, \{Q'\}$ for each path c'' in segment c'

Program segments

To express segments, need new command

$$\textbf{assume}\ A \quad \text{– assume Boolean expression } A$$

with

$$\frac{\langle b, \sigma \rangle \to \textbf{true}}{\langle \textbf{assume}\ b, \sigma \rangle \to \sigma'}$$

$$\mathrm{WP}(\textbf{assume}\ A, Q) \ = \ A \Rightarrow Q$$

A while loop with invariant I

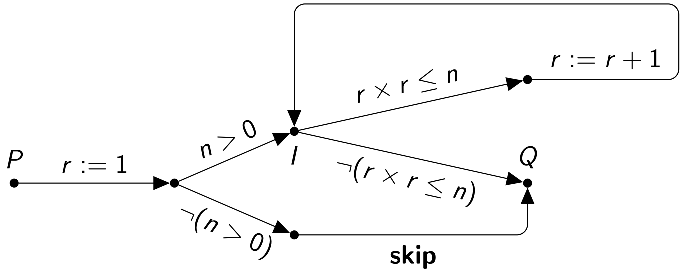
$$\{I\}\, \textbf{while}\ b\ \textbf{do}\ c$$

has

- I terminating the segment for the code before the loop
- a segment **assume** b ; c starting and ending with I.
- a segment **assume** $\neg b$ starting with I and continuing with the code after the loop

A program and its control flow graph

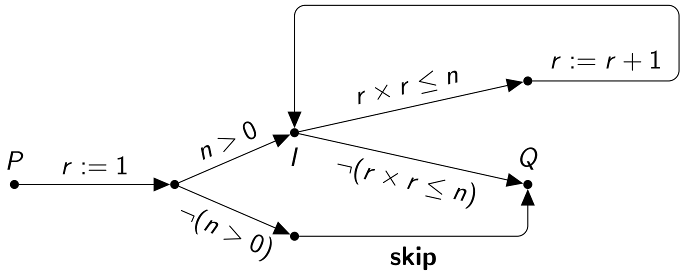
```
$\{P\}$
$r := 1$ ;
**if** $n > 0$ **then**
  $\{I\}$ **while** $r \times r \leq n$ **do** $r := r + 1$
**else**
  **skip**
$\{Q\}$
```



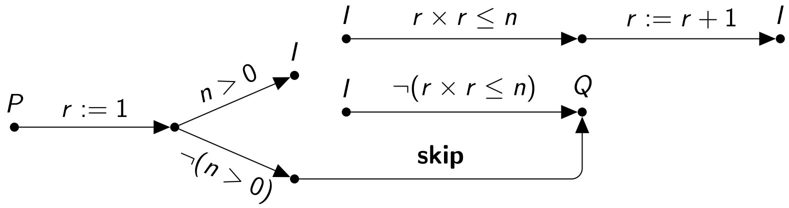
where **assume** b is abbreviated to b

Splitting control flow graph into segments

Control flow graph with cycle for loop:



Splitting at loop invariant I yields acyclic segments:



Enumerating paths of each segment

With segments:

$$\begin{tikzcd}
 & & I & I \arrow[r, "r \times r \leq n"] & \bullet \arrow[r, "r := r + 1"] & I \\
P \arrow[r, "r := 1"] & \bullet \arrow[ur, "n > 0"] \arrow[dr, "\neg(n > 0)"'] & & I \arrow[r, "\neg(r \times r \leq n)"] & Q & \\
 & & \bullet \arrow[urr, "\textbf{skip}"] & & &
\end{tikzcd}$$

the paths are:

$$\begin{tikzcd}
P \arrow[r, "r := 1"] & \bullet \arrow[r, "n > 0"] & I
\end{tikzcd}
\qquad
\begin{tikzcd}
I \arrow[r, "r \times r \leq n"] & \bullet \arrow[r, "r := r + 1"] & I
\end{tikzcd}$$

$$\begin{tikzcd}
I \arrow[r, "\neg(r \times r \leq n)"] & Q
\end{tikzcd}$$

$$\begin{tikzcd}
P \arrow[r, "r := 1"] & \bullet \arrow[r, "\neg(n > 0)"] & \bullet \arrow[r, "\textbf{skip}"] & Q
\end{tikzcd}$$

VC generation

Define two functions $\mathrm{Pre}(,)$ and $\mathrm{VC}(,)$.

$\mathrm{Pre}(c, Q)$ is like $\mathrm{WP}(c, Q)$ except it only computes $\mathrm{WP}(c, Q)$ for the start segment of c.

$$\begin{array}{rcl}
\mathrm{Pre}(\textbf{skip}, Q) & = & Q \\
\mathrm{Pre}(x := a, Q) & = & Q[x \mapsto a] \\
\mathrm{Pre}(c_0 \; ; \; c_1, Q) & = & \mathrm{Pre}(c_0, \mathrm{Pre}(c_1, Q)) \\
\mathrm{Pre}(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, Q) & = & (b \Rightarrow \mathrm{Pre}(c_0, Q)) \\
 & & \wedge \; (\neg b \Rightarrow \mathrm{Pre}(c_1, Q)) \\
\mathrm{Pre}(\{I\} \textbf{ while } b \textbf{ do } c, Q) & = & I
\end{array}$$

VC generation cont.

$\mathrm{VC}(c, Q)$ computes VCs for all but the start segment of c.

$$\begin{array}{rcl}
\mathrm{VC}(\textbf{skip}, Q) & = & \textbf{true} \\
\mathrm{VC}(x := a, Q) & = & \textbf{true} \\
\mathrm{VC}(c_0 \,;\, c_1, Q) & = & \mathrm{VC}(c_0, \mathrm{Pre}(c_1, Q)) \wedge \mathrm{VC}(c_1, Q) \\
\mathrm{VC}(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, Q) & = & \mathrm{VC}(c_0, Q) \wedge \mathrm{VC}(c_1, Q) \\
\mathrm{VC}(\{I\}\,\textbf{while } b \textbf{ do } c, Q) & = & (I \wedge b \Rightarrow \mathrm{Pre}(c, I)) \\
 & & \wedge (I \wedge \neg b \Rightarrow Q)
\end{array}$$

Soundness of VC generation

If

$$\models \forall \bar{x}.\ \ (P \Rightarrow \mathrm{Pre}(c, Q))\ \wedge\ \mathrm{VC}(c, Q)$$

then

$$\models \{P\}\, c\, \{Q\}$$

Further reading

See Concrete Semantics by Nipkow and Klein

http://www.concrete-semantics.org

- Section 7.1 on IMP language
- Section 7.2 on big-step semantics
- Section 12.4 on VC generation