Towards programming logics for low level languages

Ando Saabas

Institute of Cybernetics / INRIA Sophia-Antipolis

01.02.2004

Motivation

- In case of smart-cards, port-issuance downloading of code is possible, raising major security issues.
- Besides obvious security guarantees, some guarantees about functional properties of this code might be needed.
- Typically, developers can use interactive verification tools to get some guarantees about functional and behavioural properties of a program.
- How to bring these benefits to the code user?

Outline

- Proof carrying code.
- Source, target and assertion language.
- The weakest precondition calculus.
- Some small examples.
- Conclusion and further work.

Proof carrying code

The code developer...

- Writes a program A, annotates it with (Hoare style) specifications S, and builds a proof P that A abides to S using some verification environment.
- Compiles the program, its specification and the proof, obtaining a compiled program $\underline{A}$, a (compiled) specification $\underline{S}$, and a (compiled) proof $\underline{P}$.

The code consumer...

- Generates the set of proof obligations from $\underline{\mathrm{A}}$ and $\underline{\mathrm{S}}$ using a weakest precondition calculus.
- Uses a simple and fast proof checker to check if the proof $\underline{\mathrm{P}}$ is valid.

The source language

$e ::= x \mid n \mid e \; op \; e$

$c ::= x := e \mid \texttt{skip} \mid \texttt{if} \; e \; \texttt{then} \; c \; \texttt{else} \; c \mid c; c \mid \texttt{while} \; \{I\} \; e \; \texttt{do} \; c$

$Prog ::= \{P\} \; c \; \{Q\}$

Instruction set of the target language

i	::=	`prim` op	primitive operation
	|	`push` n	push n on stack
	|	`load` x	load value of x on stack
	|	`store` x	store top of stack in x
	|	`if` j	conditional jump
	|	`goto` j	unconditional jump

where op is a primitive operation $+, -, \times, \ldots$, or a comparison operation $<, \leq, =, \ldots$;

The assertion language

$$
\begin{array}{rcl}
P, Q & ::= & a_1 \ bop \ a_2 \\
& | & true \\
& | & P \vee Q \\
& | & P \Rightarrow Q \\
& | & P \wedge Q \\
& | & \neg P \\
& | & \exists x.P \\
& | & \forall x.P
\end{array}
$$

where $a ::= n \mid x \mid a_1 \ aop \ a_2$

The "low level" assertion language

$$
\begin{array}{rcl}
P, Q & ::= & a_1 \ bop \ a_2 \\
& | & true \\
& | & P \vee Q \\
& | & P \Rightarrow Q \\
& | & P \wedge Q \\
& | & \neg P \\
& | & \exists x.P \\
& | & \forall x.P
\end{array}
$$

where $a ::= n \mid x \mid a_1 \ aop \ a_2 \mid top \mid stack(se)$

$se ::= top \mid top - n$

Weakest precondition calculus

We have a Hoare triple - a program with a pre- and postcondition: {P} c {Q}.

To check whether the program respects the specification, calculate the weakest precondition of the program...

$$wp(c, Q) = \{s \in \Sigma_\perp \mid C[c]s \vDash Q\}$$

...and check if the precondition implies the weakest precondition

$\vDash P \Rightarrow wp(c, Q)$

Defining the weakest precondition calculus – how to deal with the unstructuredness of the assembly code?

Divide the code into blocks:

Definition 1 (Basic blocks) *Let $P[j]$ be the j-th instruction of an assembly program.*

1. *A basic block b is defined by an interval $(i, j]$ such that $P[j]$ is a jump instruction and i is the smallest program point k with $k < j$ and for all $k' \geq k$ we have that $P[k']$ is not a jump instruction.*
2. *$(i, j]$ is the successor of $(i', j']$, or equivalently $(i', j']$ is a predecessor of $(i, j]$, if $P[j'] =$ `goto` i or $P[j'] =$ `if` i.*
3. *a sub-block b' is an interval $(i, j-1)$, ie a block without its terminating jump instruction.*

Dealing with loops

- For every block, the set of its predecessors and successors can be calculated.
- In case there is a circular reference between blocks, a loop triple (l_p, l_b, l_c) can be built, ie find the loop body, the loop conditional, and the loop predecessor.
- The loop conditional has to be annotated with an invariant.

The calculus wp_s a for sub-blocks

$$wp_s(b_1; b_2, \varphi) = wp_s(b_1; wp_s(b_2, \varphi))$$

$$wp_s(\texttt{push}\ n, \varphi) = \varphi[stack(t) \leftarrow n, t \leftarrow t + 1]$$

$$wp_s(\texttt{prim}\ op, \varphi) = \varphi[stack(t) \leftarrow stack(t)\ op\ stack(t-1), top \leftarrow t - 1]$$

$$wp_s(\texttt{load}\ x, \varphi) = \varphi[stack(t) \leftarrow x, t \leftarrow t + 1]$$

$$wp_s(\texttt{store}\ x, \varphi) = \varphi[x \leftarrow stack(t), t = t - 1]$$

Example 1

```
{P} x = 2 + y {x > 5}
```

	$P \Rightarrow y > 3$
`load y`	$2 + y > 5$
`push 2`	$2 + stack(top) > 5$
`prim +`	$stack(top) + stack(top - 1) > 5$
`store` x	$stack(top) > 5$
	$x > 5$

The wp for blocks

The weakest precondition φ_i of a block b_i is ..
if b_i terminates on a

- `return`
  $\varphi_i = wp_s(b'_i, post)$
- `goto l`
  $\varphi_i = wp_s(b'_i, \varphi'_{i,l})$
- `if l`

$$\varphi_i = wp_s(b'_i, \quad stack(top) \Rightarrow \varphi'_{i,next(i)}[top \leftarrow top - 1] \wedge$$
$$\neg stack(top) \Rightarrow \varphi'_{i,l}[top \leftarrow top - 1])$$

  where..

1. $\varphi'_{i,l} = I$, if b_i is the loop body and b_l is a loop conditional
2. $\varphi'_{i,l} = I \wedge \forall y.(I \Rightarrow \varphi_l)$ if b_i is its loop predecessor and b_l is a loop conditional.
3. otherwise $\varphi'_{i,l} = \varphi_j$

Example 2

`{P} if x > 3 then y = 2 else y = 1 {y = 1}`

		$x \leq 3$
	`push 3`	$x > 3 \Rightarrow 2 = 1 \wedge x \leq 3 \Rightarrow 1 = 1$
	`load x`	$x > stack(top) \Rightarrow 2 = 1 \wedge ...$
	`prim >`	$stack(top) > stack(top-1) \Rightarrow 2 = 1 \wedge ...$
	`if 1`	$stack(top) \Rightarrow 2 = 1 \wedge \neg stack(top) \Rightarrow 1 = 1$
	`push 2`	$2 = 1$
	`store y`	$stack(top) = 1$
	`goto 2`	$y = 1$
1 :	`push 1`	$1 = 1$
	`store y`	$stack(top) = 1$
2 :		$y = 1$

Some results

Theorem 1 (Soundness of wp rules) *For all states s, programs c and assertions P $s \xrightarrow{c} s' \wedge s \vDash wp(c, P) \Rightarrow s' \vDash post$*

Theorem 2 *For all while program c and assertions P, the weakest precondition of c is equal to the weakest precondition of its compiled counterpart $\mathcal{C}(c)$*
$wp_w(c, post) = wp_a(\mathcal{C}(c), post)$

Conclusion and ongoing work.

- Defining a *wp* for Java bytecode instructions.
- Dealing with optimizations.
- An implementation.