Program AlsoInteresting

```plaintext
while read() != 0
  i := 0
  while i < 100
    use 1
    i := i + 1
```

Some loose ends

“Certified code is an old idea”
Defining a VC\text{gen}

To define a verification-condition generator for our language, we start by defining the language of predicates:

\[
P ::= b \\
| P \land P \\
| A \land P \\
| e? P : P
\]

\[
A ::= b \\
| A \land A
\]

\[
b ::= \text{true} \\
| \text{false} \\
| e \land e \\
| e = e
\]

\[\text{annotations}\]

\[\text{predicates}\]

\[\text{boolean expressions}\]

Weakest preconditions

The VC\text{gen} we define is a simple variant of Dijkstra’s weakest precondition calculus.

It makes use of generalized predicates of the form: \((P, e)\)

- \((P, e)\) is true if \(P\) is true and at least \(e\) units of the resource are currently available

Hoare triples

The VC\text{gen}’s job is to compute, for each statement \(S\) in the program, the Hoare triple

\[\text{**}(P’, e’)\text{\ }S\text{\ }\text{**(P, e)}\]

which means, roughly:

- If \((P, e)\) holds prior to executing \(S\), and then \(S\) is executed and it terminates, then \((P’, e’)\) holds afterwards

VC\text{gen}

Since we will usually have the postcondition (true, 0) for the last statement in the program, we can define a function

\[\text{vcg}(S, (P, i)) : (P’, i’)\]

I.e., given a statement and its postcondition, generate the weakest precondition

The VC\text{gen} (easy parts)

\[
\text{vcg}(\text{skip}, (P, e)) = (P, e)
\]

\[
\text{vcg}(s_1; s_2, (P, e)) = \text{vcg}(s_1, \text{vcg}(s_2, (P, e)))
\]

\[
\text{vcg}(x := e’, (P, e)) = ([e’/x]P, [e’/x]e)
\]

\[
\text{vcg}(\text{if } b \text{ then } s_1 \text{ else } s_2, (P, e)) =
\]

\[
(b? P_1; P_2, b? e_1; e_2)
\]

where \((P_1, e_1) = \text{vcg}(s_1, (P, e))\)

and \((P_2, e_2) = \text{vcg}(s_2, (P, e))\)

\[
\text{vcg}(\text{use } e’, (P, e)) = (P \land e’, 0,
\]

\[
e’ + (e, 0? e : 0)
\]

\[
\text{vcg(\text{acquire } e’, (P, e)) = (P \land e’, 0, e-e’)}
\]

Example 1

Prove: \(\text{Pre} (\text{true}, -1)\)

\[
\text{Pre: (true, 0)}
\]

\[
\text{acquire 3}
\]

\[
\text{use 2}
\]

\[
\text{Post: (true, 0)}
\]

\[
\text{vcg(use } e’, (P, e)) = (P \land e’, 0, e’ + (e, 0? e : 0))
\]

\[
\text{vcg(acquire } e’, (P, e)) = (P \land e’, 0, e-e’)
\]
Example 2

acquire 3
use 2
use 1

\[
\text{vcg}(\text{use } e', (P, e)) = (P \land e' \land 0, e' + (e \land 0)^{e'})
\]

\[
\text{vcg}(\text{acquire } e', (P, e)) = (P \land e' \land 0, e - e')
\]

Example 3

acquire 9
if (b)
then use 5
else use 4
use 4

\[
\text{vcg}(\text{if } b \text{ then } s1 \text{ else } s2, (P, e)) = (b? P1:P2, b? e1:e2)
\]

where (P1, e1) = vcg(s1, (P, e))
and (P2, e2) = vcg(s2, (P, e))

Example 4

acquire 8
if (b)
then use 5
else use 4
use 4

\[
\text{vcg}(\text{if } b \text{ then } s1 \text{ else } s2, (P, e)) = (b? P1:P2, b? e1:e2)
\]

where (P1, e1) = vcg(s1, (P, e))
and (P2, e2) = vcg(s2, (P, e))

Loops

Loops cause an obvious problem for the computation of weakest preconditions

acquire n
i := 0
while (i<n) do {
use 1
i := i + 1
}

Snipping up programs

A simple loop

Broken into segments

Snip invariants

We thus require that the programmer or compiler insert invariants to cut the loops

acquire n
i := 0
while (i<n) do {
use 1
i := i + 1
} with (i, n-i)

An annotated loop
### VCgen for loops

\[
\text{vcg}(\text{while } b \text{ do } s \text{ with } (A_1, e_1), (P, e)) = \\
(A_2 \land s_1, s_2, \ldots, a_2) \text{ if } P \land e_1, e_2, \\
\text{where } (P', e') = \text{vcg}(s, (A_2, e_2)) \\
\text{and } i_1, i_2, \ldots \text{ are the variables modified in } s
\]

### Example 5

\[
\begin{align*}
\text{acquire } n; \\
i &:= 0; \\
\text{while } (i < n) \text{ do } \\
&\quad \text{use } 1; \\
&\quad i := i + 1; \\
&\quad \text{with } (i \cdot n, n - i); \\
\text{with } (i \cdot n, n - i); \\
\end{align*}
\]

### Our easy case

**Program Static**

```plaintext
acquire 10000  \\
i := 0  \\
while i < 10000  \\
use 1  \\
i := i + 1  \\
with (i \cdot 10000, 10000 - i)
```

**Typical loop invariant for "standard for loops"**

### Our hopeless case

**Program Dynamic**

```plaintext
while read() != 0  \\
acquire 1  \\
use 1  \\
with (true, 0)
```

**Typical loop invariant for "Java-style checking"**

### Our interesting case

**Program Interesting**

```plaintext
N := read()  \\
acquire N  \\
i := 0  \\
while i < N  \\
use 1  \\
i := i + 1  \\
with (i \cdot N, N - i)
```

### Also interesting

**Program AlsoInteresting**

```plaintext
while read() != 0  \\
acquire 100  \\
i := 0  \\
while i < 100  \\
use 1  \\
i := i + 1  \\
with (i \cdot 100, 100 - i)
```
Annotating programs

How are these annotations to be inserted?

- The programmer could do it
- A compiler could start with code that has every use immediately preceded by an acquire
- We then have a code-motion optimization problem to solve

VC Explosion

Exponential growth in size of the VC is possible.

Proving the Predicates

Proving predicates

Note that left-hand side of implications is restricted to annotations

- \( \text{vcg()} \) respects this, as long as loop invariants are restricted to annotations

\[
\begin{align*}
P & ::= b \\
 & | P \land P \\
 & | A \land P \\
& | \text{Si}.P \\
& | e? P : P
\end{align*}
\]

\[
\begin{align*}
A & ::= b \\
 & | A \land A
\end{align*}
\]

boolean expressions

\[
\begin{align*}
b & ::= \text{true} \\
 & | \text{false} \\
& | \text{e, e} \\
& | \text{e = e}
\end{align*}
\]

predicates

VCGen’s Complexity

Some complications:

- If dealing with machine code, then VCGen must parse machine code.
- Maintaining the assumptions and current context in a memory-efficient manner is not easy.

Note that Sun’s kVM does verification in a single pass and only 8KB RAM!
A simple prover

We can thus use a simple prover with functionality

• prove(annot, pred) : bool

where prove(A,P) is true iff A |- P

• i.e., A |- P holds for all values of the variables introduced by \$

prove(A,b) = :sat(A \&\& :b)
prove(A,p_1 \&\& p_2) = prove(A,p_1) \&\& prove(A,p_2)
prove(A,b? p_1:p_2) = prove(A \&\& b,p_1) \&\&
prove(A \&\& :b,p_2)
prove(A,A_1 \&\& p) = prove(A \&\& A_1,p)
prove(A,[a/b]p) = prove(A,[a/b]p) (a fresh)

Soundness

Soundness is stated in terms of a formal operational semantics.

Essentially, it states that if

• Pre \&\& vcg(program)

holds, then all use e statements succeed

Logical Frameworks

Logical frameworks

The Edinburgh Logical Framework (LF) is a language for specifying logics.

Kinds K ::= Type \&\& \Pi x : A.K
Types A ::= a \&\& A.M \&\& \Pi x : A_1.A_2
Objects M ::= x \&\& c \&\& M_1M_2 \&\& \lambda x : A.M

LF is a lambda calculus with dependent types, and a powerful language for writing formal proof systems.

LF

The Edinburgh Logical Framework language, or LF, provides an expressive language for proofs-as-programs.

Furthermore, it use of dependent types allows, among other things, the axioms and rules of inference to be specified as well.
Pfenning's Elf

Several researchers have developed logic programming languages based on these principles.

One of special interest, as it is based on LF, is Pfenning's Elf language and system.

```
true : pred.
false : pred.
\& : pred -> pred -> pred.
\| : pred -> pred -> pred.
=> : pred -> pred -> pred.
all : (exp -> pred) -> pred.
```

This small example defines the abstract syntax of a small language of predicates.

Elf example

So, for example:

\[ \forall A, B. A \land B \Rightarrow B \land A \]

Can be written in Elf as

```
all([a:pred] all([b:pred] => (\& a b) (\& b a)))
```

Proofs in Elf

...which in turn allows us to have easy-to-validate proofs

```
... (impi (\& a b) (\& b a) (andl a b a (ander a b ab) (andr a b ab))...) : all([a:exp] all([b:exp] => (\& a b) (\& b a))).
```

LF as the internal language

```
pf : pred -> type.
truei : pf true.
andi : (P:pred) (Q:pred) pf P -> pf Q -> pf (\& P Q).
andel : (P:pred) (Q:pred) pf (\| P Q) -> pf P.
ander : (P:pred) (Q:pred) pf (\| P Q) -> pf Q.
impi : (P1:pred) (P2:pred) (pf P1 -> pf P2) -> pf (impi P1 P2).
alli : (P1:exp -> pred) ((X:exp) pf (P1 X)) -> pf (all P1).
```

Code producer

Host
This store instruction is dangerous!

A verification condition

I am convinced it is safe to execute only if \( (\forall a \exists a') (\forall b \exists b') (a = a' \land b = b') \)

\( (\forall a \exists a') (\forall b \exists b') (a = a' \land b = b') \)

Your proof typechecks. I believe you because I believe in logic.