Back to our case study

Program AlsoInteresting

while read() != 0

    i := 0

while i < 100

    use 1

    i := i + 1
The language

\[ s ::= \text{skip} \]
\[ \quad | \quad i := e \]
\[ \quad | \quad \text{if e then s else s} \]
\[ \quad | \quad \text{while e do s} \]
\[ \quad | \quad s ; s \]
\[ \quad | \quad \text{use e} \]
\[ \quad | \quad \text{acquire e} \]
Defining a VCgen

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

\[
P ::= b \\
  \mid P \land P \\
  \mid A \implies P \\
  \mid \forall i. P \\
  \mid e? P : P
\]

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

Annotations

\[
b ::= true \\
  \mid false \\
  \mid e \geq e \\
  \mid e = e
\]

Boolean expressions

Predicates
Weakest preconditions

The VCgen 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 VCgen's job is to compute, for each statement $S$ in the program, the Hoare triple

$$\textbullet\quad (\mathcal{P}', e') \ S \ (\mathcal{P}, e)$$

which means, roughly:

$$\textbullet\quad \text{If } (\mathcal{P}, e) \text{ holds prior to executing } S, \text{ and then } S \text{ is executed and it terminates, then } (\mathcal{P}', e') \text{ holds afterwards}$$
Since we will usually have the postcondition \((true, 0)\) for the last statement in the program, we can define a function

\[ \text{vcg}(S, (P, i)) \rightarrow (P', i') \]

I.e., given a statement and its postcondition, generate the weakest precondition
The VCgen (easy parts)

\[
\begin{align*}
\text{vcg}(\text{skip}, (P,e)) &= (P,e) \\
\text{vcg}(s_1;s_2, (P,e)) &= \text{vcg}(s_1, \text{vcg}(s_2, (P,e))) \\
\text{vcg}(\text{x:=e'}, (P,e)) &= ([e'/x]P, [e'/x]e) \\
\text{vcg}(\text{if b then } s_1 \text{ else } s_2, (P,e)) &= \\
&\qquad (b? P_1:P_2, b? e_1:e_2) \\
&\quad \text{where } (P_1,e_1) = \text{vcg}(s_1,(P,e)) \\
&\quad \text{and } (P_2,e_2) = \text{vcg}(s_2,(P,e)) \\
\text{vcg}(\text{use e'}, (P,e)) &= (P \land e'\geq 0, \\
&\quad e' + (e\geq 0? e : 0) \\
\text{vcg}(\text{acquire e'}, (P,e)) &= (P \land e'\geq 0, e-e')
\end{align*}
\]
Example 1

Prove: Pre $\Rightarrow$ (true, -1)

Pre: (true, 0)

- acquire 3
- use 2

Post: (true, 0)

$(true \land 2 \geq 0 \land 3 \geq 0, 2+0-3)$

$(true \land 2 \geq 0, 2+0)$

$(true, 0)$

$vCG(use\ e', (P,e)) = (P \land e' \geq 0, e' + (e \geq 0? e:0))$

$vCG(acquire\ e', (P,e)) = (P \land e' \geq 0, e-e')$
Example 2

\[
\begin{align*}
\text{acquire} & 3 \\
\text{use} & 2 \\
\text{use} & 1
\end{align*}
\]

\[
(\text{true} \land 1 \geq 0 \land 2 \geq 0 \land 3 \geq 0, 2+1+0-3)
\]

\[
(\text{true} \land 1 \geq 0 \land 2 \geq 0, 2+1+0)
\]

\[
(\text{true} \land 1 \geq 0, 1+0)
\]

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

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

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

Example 3

```plaintext
acquire 9
if (b)	hen use 5
else use 4
use 4
```

\[(9 \geq 0, (b \? 9:8) - 9)\]
\[(b \? \text{true: true}, b \? 9:8)\]
\[(5 \geq 0, 9)\]
\[(4 \geq 0, 8)\]
\[(4 \geq 0, 4)\]
\[(\text{true}, 0)\]

\[\text{vcg}(\text{if } b \text{ then } s1 \text{ else } s2, (P,e)) = \]
\[(b \? P1:P2, b \? e1:e2)\]
\[\text{where } (P1,e1) = \text{vcg}(s1,(P,e))\]
\[\text{and } (P2,e2) = \text{vcg}(s2,(P,e))\]
Example 4

\[ \text{acquire 8} \]
\[ \text{if (b)} \]
\[ \quad \text{then use 5} \]
\[ \quad \text{else use 4} \]
\[ \text{use 4} \]

\[ (8 \geq 0, (b?9:8) - 8) \]
\[ (b?\text{true}:\text{true}, b?9:8) \]
\[ (5 \geq 0, 9) \]
\[ (4 \geq 0, 8) \]
\[ (4 \geq 0, 4) \]
\[ (\text{true}, 0) \]

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

Pre

\[I\]

Post

Broken into segments

Pre

\[I\]

\[I\]

\[I\]

Post
Loop 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
}
```

An annotated loop

```
A ::= b
    | A ∧ A
```

An annotated loop
vcg(while b do s with (Aᵢ,eᵢ), (P,e)) =
(Aᵢ ∧ ∀i₁,i₂,....Aᵢ ⇒ b ? P’ ∧ eᵢ≥e’,
  : P ∧ eᵢ≥e,
  eᵢ)

where (P’,e’) = vcg(s,(Aᵢ,eᵢ))

and i₁,i₂,... are the variables modified in s
Example 5

acquire n;
i := 0;

while (i<n) do {
    use 1;
i := i + 1;
} with (i≤n, n-i);
(... \land n≥0, n-n)
(0≤n \land \forall i. ..., n-0)
(i≤n \land \forall i. i≤n \Rightarrow
cond(i<n, i+1≤n \land n-i≥n-i, n-i≥n-i)
    n-i)
    (i+1≤n \land 1≥0, n-i)
    (i+1≤n, n-(i+1))
    (i≤n, n-i)
    (true, 0)
Our easy case

Program Static
acquire 10000
i := 0
while i < 10000
  use 1
  i := i + 1
with (i ≤ 10000, 10000 - i)

Typical loop invariant for “standard for loops”
Our hopeless case

Program Dynamic
while read() != 0
    acquire 1
    use 1
with (true, 0)

Typical loop invariant for “Java-style checking”
Our interesting case

Program Interesting

N := read()
acquire N
i := 0
while i < N
    use 1
    i := i + 1
with (i≤N, N-i)
Also interesting

Program AlsoInteresting
    while read() != 0
        acquire 100
        i := 0
        while i < 100
            use 1
            i := i + 1
        with (i \leq 100, 100-i)
Annotating programs

How are these annotations to be inserted?

• The programmer could do it

Or:

• 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
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!
VC Explosion

\[
a = b \implies (x = c \implies \text{safe}_f(y, c)) \land \\
\quad x \not\equiv c \implies \text{safe}_f(x, y)
\]
\[
\land \\
a \not\equiv b \implies (a = x \implies \text{safe}_f(y, x)) \land \\
\quad a \not\equiv x \implies \text{safe}_f(a, y)
\]

Exponential growth in size of the VC is possible.
(a=b \implies P(x,b,c,x) \land 
 a<>b \implies P(a,b,x,x))
\land 
(
\forall a',c'. \ P(a',b,c',x) \implies 
 a'=c' \implies safe_{\bar{f}}(y,c') \land 
 a'<>c' \implies safe_{\bar{f}}(a',y)
)

Growth can usually be controlled by careful placement of just the right “join-point” invariants.
Proving the Predicates
Proving predicates

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

- `vcg()` respects this, as long as loop invariants are restricted to annotations

```
P ::= b
| P ∧ P
| A ⇒ P
| ∀i.P
| e? P : P
```

```
A ::= b
| A ∧ A
```

```
b ::= true
| false
| e ≥ e
| e = e
```

predicates

annotations

boolean expressions
A simple prover

We can thus use a simple prover with functionality

- `prove(annotation,pred) → bool`

where `prove(A,P)` is true iff `A \Rightarrow P`

- i.e., `A \Rightarrow P` holds for all values of the variables introduced by `\forall`
A simple prover

\[
\begin{align*}
\text{prove}(A,b) & = \neg \text{sat}(A \land \neg b) \\
\text{prove}(A,P_1 \land P_2) & = \text{prove}(A,P_1) \land \text{prove}(A,P_2) \\
\text{prove}(A,b? P_1:P_2) & = \text{prove}(A \land b,P_1) \land \text{prove}(A \land \neg b,P_2) \\
\text{prove}(A,A_1 \Rightarrow P) & = \text{prove}(A \land A_1,P) \\
\text{prove}(A,\forall i.P) & = \text{prove}(A,[a/i]P) \text{ (a fresh)}
\end{align*}
\]
Soundness

Soundness is stated in terms of a formal operational semantics.

Essentially, it states that if

- $\text{Pre} \Rightarrow \text{vcg}(\text{program})$

holds, then all $\textbf{use e}$ statements succeed
Logical Frameworks
The Edinburgh Logical Framework (LF) is a language for specifying logics.

Kinds \( K \ ::= \) Type | \( \Pi x : A.K \)

Types \( A \ ::= a \mid A.M \mid \Pi x : A_1.A_2 \)

Objects \( M \ ::= x \mid c \mid M_1.M_2 \mid \lambda x : A.M \)

LF is a lambda calculus with dependent types, and a powerful language for writing *formal proof systems*. 
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.
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.
\(\land\) : pred -> pred -> pred.
\(\lor\) : pred -> pred -> pred.
\(\Rightarrow\) : 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

\[
\text{all}([a:\text{pred}] \text{ all}([b:\text{pred}]
    
    => (\land a b) (\land b a))
\]

<table>
<thead>
<tr>
<th>true</th>
<th>: pred.</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>: pred.</td>
</tr>
<tr>
<td>\land</td>
<td>: pred \to pred \to pred.</td>
</tr>
<tr>
<td>\lor</td>
<td>: pred \to pred \to pred.</td>
</tr>
<tr>
<td>=&gt;</td>
<td>: pred \to pred \to pred.</td>
</tr>
<tr>
<td>all</td>
<td>: (exp \to pred) \to pred.</td>
</tr>
</tbody>
</table>
Dependent types allow us to define the proof rules...

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 (=> P1 P2).
alli : {P1:exp -> pred} ({X:exp} pf (P1 X)) -> pf (all P1).
e : exp -> pred
Proofs in Elf

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

... (impi (/\ a b) (/\ b a)
    ([ab:pf(/\ a b)]
     (andi (ander ab)
      (andel ab)))) ...):

all([a:exp] all([b:exp]
    => (/\ a b) (/\ b a))).
LF as the internal language

Agent

Code

Explanation

Verification condition generator

Checker

Proof rules

Host

*LF is the language of the blue arrows*
Code producer

Host
This *store* instruction is dangerous!
I am convinced it is safe to execute only if all([a:exp] (all([b:exp] (=> (\ a b) (\ b a)))))
... (impi (\ a b) (\ b a)
  ([ab:pf(\ a b)]
   (andi b a (ander a b ab)
    (andel a b ab)))))...)

Code producer                       Host
Your proof typechecks. I believe you because I believe in logic.