Lectures on Proof-Carrying Code

Peter Lee

Carnegie Mellon University

Lecture 2 (of 3)
June 21-22, 2003
University of Oregon

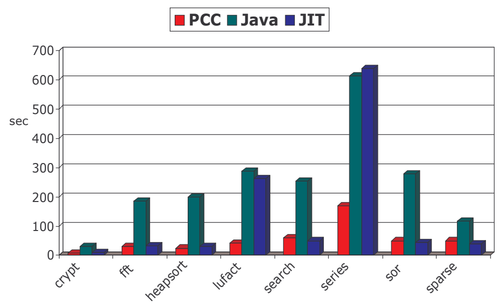
2004 Summer School on Software Security

Some loose ends

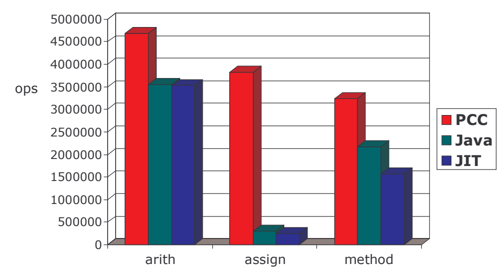
"Certified code is an old idea"

- see Butler Lampson's 1974 paper: An open operating system for a single-user machine. *Operating Systems Proceedings of an International Symposium*, LNCS 16.

Java Grande Suite



Java Grande Benchmark Suite



Back to our case study

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

The language

```
s ::= skip
    | i := e
    | if e then s else s
    | while e do s
    | s ; s
    | use e
    | acquire e
```

Defining a VCgen

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

```
P ::= b
    | P Æ P
    | A ) P
    | 8i.P
    | e? P : P
```

predicates

```
A ::= b
    | A Æ A
```

annotations

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

boolean expressions

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

- **(P',e') S (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

VCgen

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

- vcg(S, (P,i)) ! (P',i')

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

The VCgen (easy parts)

```
vcg(skip, (P,e))      = (P,e)
vcg(s1;s2, (P,e))     = vcg(s1, vcg(s2, (P,e)))
vcg(x:=e', (P,e))     = ([e'/x]P, [e'/x]e)
vcg(if b then s1 else s2, (P,e)) =
          (b? P1:P2, b? e1:e2)
              where (P1,e1) = vcg(s1,(P,e))
              and   (P2,e2) = vcg(s2,(P,e))
vcg(use e', (P,e))    = (P Æ e'¸0,
                         e' + (e¸0? e : 0)
vcg(acquire e', (P,e)) = (P Æ e'¸0, e-e')
```

Example 1

Prove: Pre) (true,-1)

Pre: (true,0)

(true Æ 2¸0 Æ 3¸0, 2+0-3)

```
acquire 3
use 2
```

(true Æ 2¸0, 2+0)

Post: (true,0)

(true, 0)

```
vcg(use e', (P,e))     = (P Æ e'¸0, e' + (e¸0? e:0)
vcg(acquire e', (P,e)) = (P Æ e'¸0, e-e')
```

Example 2

```
acquire 3
use 2
use 1
```

(true Æ 1¸0 Æ 2¸0 Æ 3¸0, 2+1+0-3)
(true Æ 1¸0 Æ 2¸0, 2+1+0)
(true Æ 1¸0, 1+0)
(true, 0)

```
vcg(use e', (P,e))     = (P Æ e'¸0, e' + (e¸0? e:0)
vcg(acquire e', (P,e)) = (P Æ e'¸0, e-e')
```

Example 3

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

(9¸0, (b?9:8) - 9)
(b?true:true, b?9:8)
(5¸0, 9)
(4¸0, 8)
(4¸0, 4)
(true, 0)

```
vcg(if b then s1 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
```

(8¸0, (b?9:8) - 8)
(b?true:true, b?9:8)
(5¸0, 9)
(4¸0, 8)
(4¸0, 4)
(true, 0)

```
vcg(if b then s1 else s2, (P,e)) =
  (b? P1:P2, b? e1:e2)
      where (P1,e1) = vcg(s1,(P,e))
      and   (P2,e2) = vcg(s2,(P,e))
```

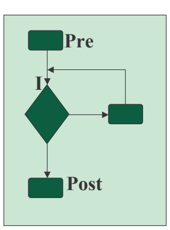
Loops

Loops cause an obvious problem for the computation of weakest preconditions

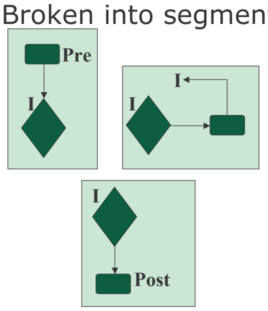
```
acquire n
i := 0
while (i<n) do {
  use 1
  i := i + 1
}
```

Snipping up programs

A simple loop



Broken into segments



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
} with (i·n, n-i)
```

An annotated loop

```
A ::= b
    | A Æ A
```

VCgen for loops

```
vcg(while b do s with (A_I,e_I), (P,e)) =
  (A_I Æ 8i_1,i_2,….A_I ) b ? P' Æ e_I¸e',
                          : P Æ e_i¸e,
   e_I)
```

where `(P',e') = vcg(s,(A_I,e_I))`

and `i_1,i_2,…` *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);
```

(… \and n¸0, n-n)

(0·n Æ 8i. …, n-0)

(i·n Æ 8i.i·n)
cond(i<n,i+1·n Æ n-i¸n-i,
n-i¸n-i)
n-i)

(i+1·n Æ 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·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

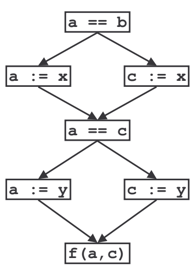
VCGen's Complexity

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!

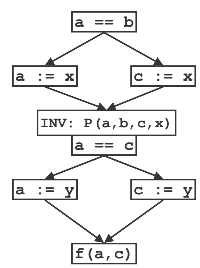
VC Explosion




```
a=b   => (x=c   => safe_f(y,c) ∧
          x<>c  => safe_f(x,y))
∧
a<>b  => (a=x   => safe_f(y,x) ∧
          a<>x  => safe_f(a,y))
```

Exponential growth in size of the VC is possible.

VC Explosion




```
(a=b   => P(x,b,c,x) ∧
 a<>b  => P(a,b,x,x))
∧
(∀a',c'. P(a',b,c',x) =>
         a'=c'   => safe_f(y,c') ∧
         a'<>c'  => safe_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 Æ P
    | A ) P
    | 8i.P
    | e? P : P
```

predicates

```
A ::= b
    | A Æ A
```

annotations

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

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) P

- i.e., A) P holds for all values of the variables introduced by 8

A simple prover

```
prove(A,b)            = :sat(A Æ :b)
prove(A,P1 Æ P2)      = prove(A,P1) Æ prove(A,P2)
prove(A,b? P1:P2)     = prove(A Æ b,P1) Æ
                        prove(A Æ :b,P2)
prove(A,A1 ) P) = prove(A Æ A1,P)
prove(A,8i.P)         = prove(A,[a/i]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	$::=$	$\mathtt{Type} \mid \Pi x : A.K$
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*.

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:

$$\forall A, B.\ A \wedge B \Rightarrow B \wedge A$$

Can be written in Elf as

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

```
true   : pred.
false  : pred.

/\     : pred -> pred -> pred.
\/     : pred -> pred -> pred.
=>     : pred -> pred -> pred.
all    : (exp -> pred) -> pred.
```

Proof rules in Elf

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
```

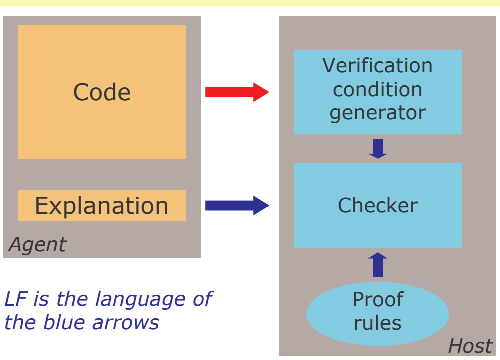
Proofs in Elf

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

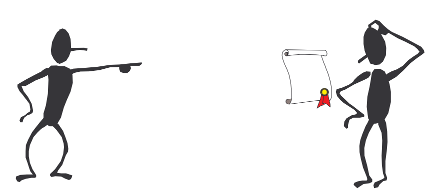
```
… (impi (/\ a b) (/\ b a)
    ([ab:pf(/\ a b)]
      (andi b a (ander a b ab)
                (andel a b ab))))…) :

all([a:exp] all([b:exp]
  => (/\ a b) (/\ b a))).
```

LF as the internal language

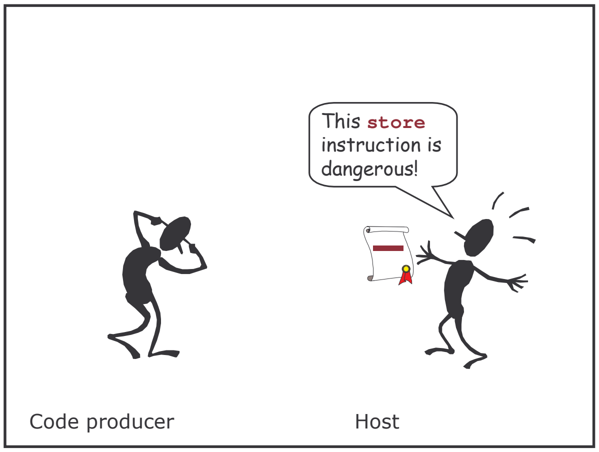


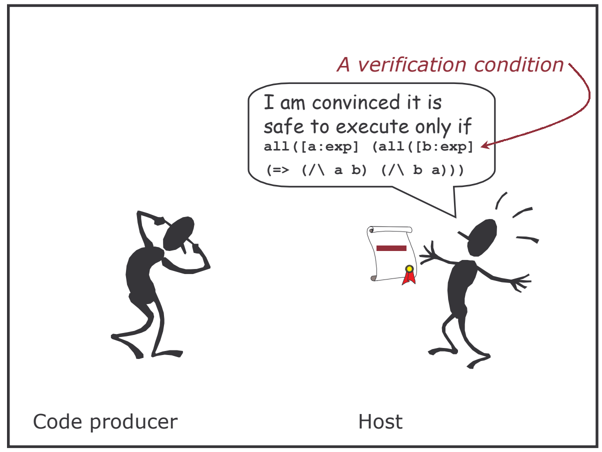
LF is the language of the blue arrows

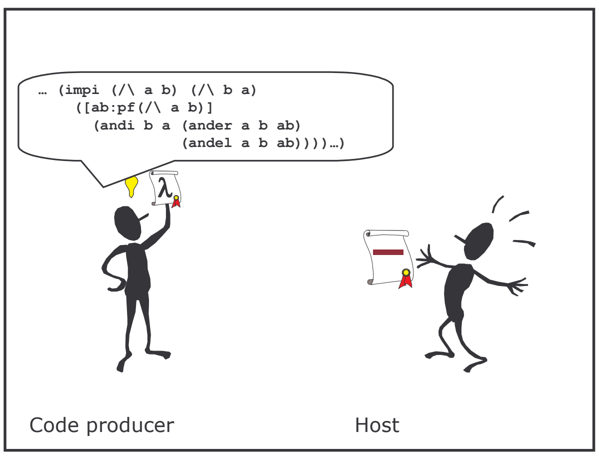


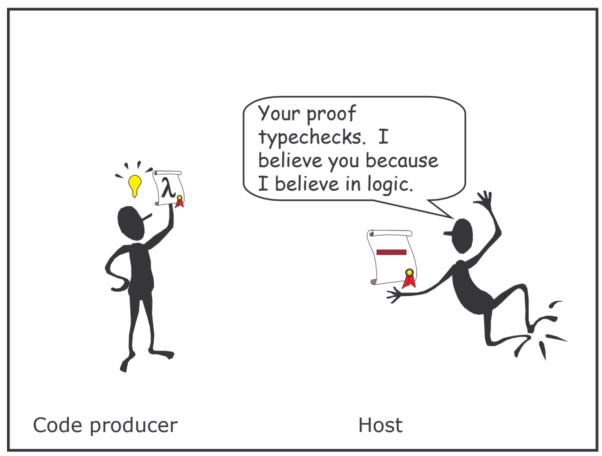
Code producer

Host


This store instruction is dangerous!
Code producer
Host


A verification condition
I am convinced it is safe to execute only if
all([a:exp] (all([b:exp]
(=> (/\ a b) (/\ b a)))
Code producer
Host


… (impi (/\ a b) (/\ b a)
([ab:pf(/\ a b)]
(andi b a (ander a b ab)
(andel a b ab))))…)
λ
Code producer
Host


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