Lectures on Proof-Carrying Code

Peter Lee

Carnegie Mellon University

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

Acknowledgments

George Necula

Frank Pfenning

Karl Crary

Zhong Shao

Bob Harper

Recap

Yesterday we

- formulated a certification problem
- defined a VCgen
  - this necessitated the use of (untrusted) loop invariant annotations
- showed a simple prover
- briefly discussed LF as a representation language for predicates and proofs

Continuing…

Today we continue by describing how to obtain the annotated programs via certifying compilation

An example of certifying compilation

```
public class Bcopy {
  public static void bcopy(int[] src,
                           int[] dst)
  {
     int l = src.length;
     int i = 0;

     for(i=0; i<l; i++) {
       dst[i] = src[i];
     }
  }
}
```

Proof rules
(excerpts)

1. Standard syntax and rules for first-order logic.

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

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.

…
```

Syntax of predicates.

Type of valid proofs, indexed by predicate.

Inference rules.

Proof rules
(excerpts)

2. Syntax and rules for arithmetic and equality.

“csuble” means ≤ in the x86 machine.

```
=       : exp -> exp -> pred.
<>      : exp -> exp -> pred.

eq_le   : {E:exp} {E':exp} pf (csubeq  E E') ->
           pf (csuble  E E').

moddist+: {E:exp} {E':exp} {D:exp}
           pf (= (mod (+ E E') D) (mod (+ (mod E D) E') D)).

=sym    : {E:exp} {E':exp} pf (=  E E') -> pf (=  E' E).
<>sym   : {E:exp} {E':exp} pf (<> E E') -> pf (<> E' E).

=tr     : {E:exp} {E':exp} {E'':exp}
           pf (= E E') -> pf (= E' E'') -> pf (= E E'').
```

Proof rules for arithmetic

Note that we avoid the need for a sophisticated decision procedure for a fragment of integer arithmetic

Intuitively, the prover only needs to be as “smart” as the compiler

Arithmetic

Note also that the “safety critical” arithmetic (i.e., array-element address computations) generated by typical compilers is simple and highly structured

- e.g., multiplications only by 2, 4, or 8

Human programmers, on the other hand, may require much more sophisticated theorem proving

Proof rules
(excerpts)

3. Syntax and rules for the Java type system.

```
jint    : exp.
jfloat  : exp.
jarray  : exp -> exp.
jinstof : exp -> exp.

of     : exp -> exp -> pred.

faddf  : {E:exp} {E':exp}
            pf (of E jfloat) ->
            pf (of E' jfloat) ->
            pf (of (fadd E E') jfloat).

ext    : {E:exp} {C:exp} {D:exp}
            pf (jextends C D) ->
            pf (of E (jinstof C)) ->
            pf (of E (jinstof D)).
```

Java typing rules in the TCB

It seems unfortunate to have Java types here, since we are proving properties of x86 machine code

More to say about this shortly…

Proof rules
(excerpts)

4. Rules describing the layout of data structures.

```
aidxi  : {I:exp} {LEN:exp} {SIZE:exp}
           pf (below I LEN) ->
           pf (arridx (add (imul I SIZE) 8) SIZE LEN).

wrArray4: {M:exp} {A:exp} {T:exp} {OFF:exp} {E:exp}
           pf (of A (jarray T)) ->
           pf (of M mem) ->
           pf (nonnull A) ->
           pf (size T 4) ->
           pf (arridx OFF 4 (sel4 M (add A 4))) ->
           pf (of E T) ->
           pf (safewr4 (add A OFF) E).
```

This “sel4” means the result of reading 4 bytes from heap M at address A+4.

Compiling model rules in the TCB

It is even more unfortunate to have rules that are specific to a single compiler here

Though it does tend to lead to compact proofs

More to say about this shortly…

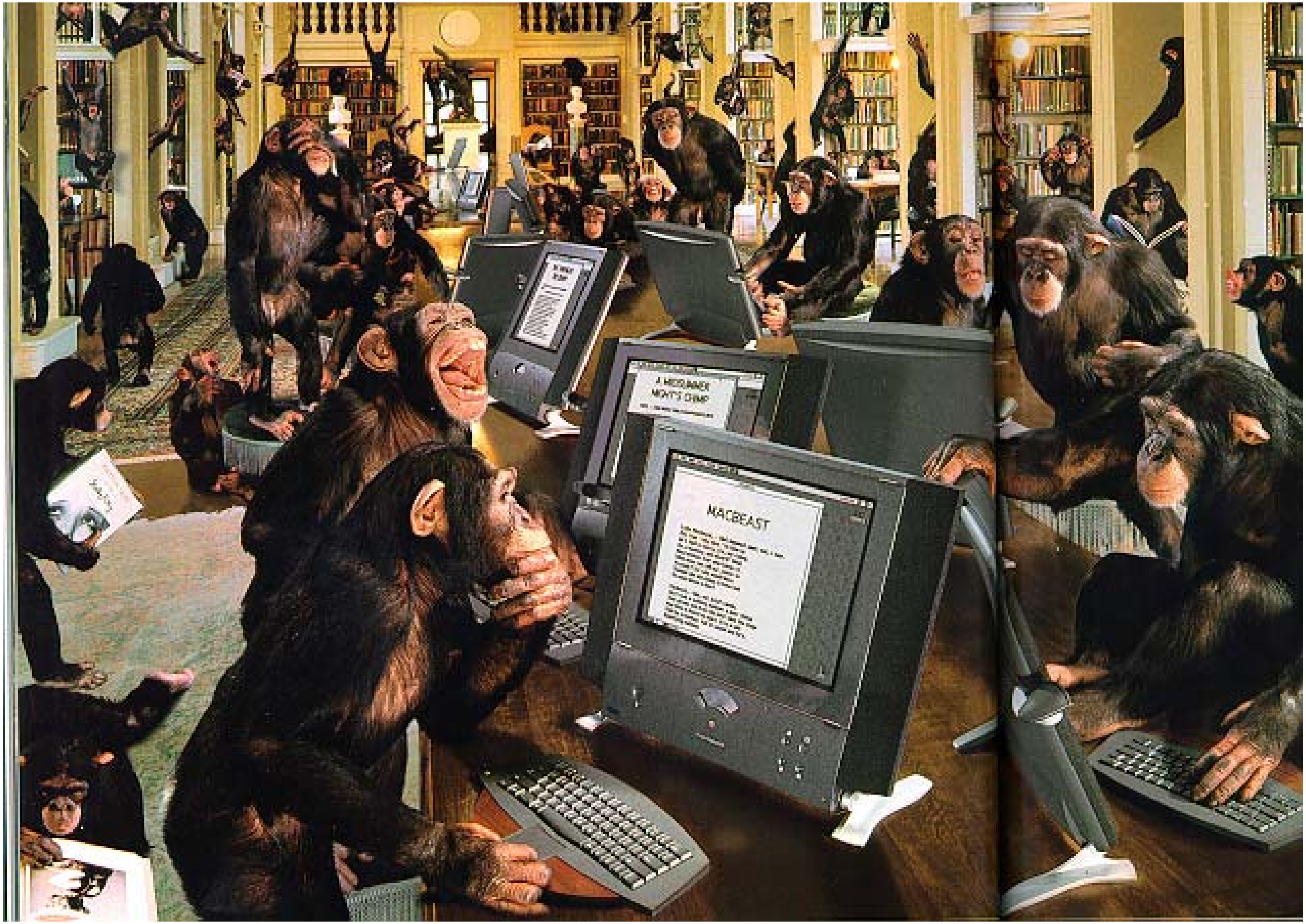
Proof rules
(excerpts)

5. Quick hacks.

```
nlt0_0 : pf (csubnlt 0 0).
nlt1_0 : pf (csubnlt 1 0).
nlt2_0 : pf (csubnlt 2 0).
nlt3_0 : pf (csubnlt 3 0).
nlt4_0 : pf (csubnlt 4 0).
```

Inevitably, “unclean” things are sometimes put into the specification…

How do we know that it is right?



Back to our example

```
public class Bcopy {
  public static void bcopy(int[] src,
                           int[] dst)
  {
     int l = src.length;
     int i = 0;

     for(i=0; i<l; i++) {
       dst[i] = src[i];
     }
  }
}
```

Unoptimized loop body

```
L11 :
        movl    4(%ebx), %eax
        cmpl    %eax, %edx
        jae     L24
L17 :
        cmpl    $0, 12(%ebp)
        movl    8(%ebx, %edx, 4), %esi
        je      L21
L20 :
        movl    12(%ebp), %edi
        movl    4(%edi), %eax
        cmpl    %eax, %edx
        jae     L24
L23 :
        movl    %esi, 8(%edi, %edx, 4)
        movl    %edi, 12(%ebp)
        incl    %edx
L9 :
ANN_INV(ANN_DOM_LOOP,
        %LF_(/\ (of rm mem ) (of loc1 (jarray jint) ))%_LF,
        RB(EBP,EBX,ECX,ESP,FTOP,LOC4,LOC3))
        cmpl    %ecx, %edx
        jl      L11
```

Bounds check on src.

Bounds check on dst.

Note: L24 raises the ArrayIndex exception.

Stack Slots

Each procedure will want to use the stack for local storage.

This raises a serious problem because a lot of information is lost by VCGen (such as the value) when data is stored into memory.

We avoid this problem by assuming that procedures use up to 256 words of stack as registers.

Unoptimized code is easy

As we saw previously in the sample program **Dynamic**, in the absence of optimizations, proving the safety of array accesses is relatively easy

Indeed, in this case it is reasonable for VCgen to verify the safety of the array accesses

Optimized target code

```
ANN_LOCALS(_bcopy__6arrays5BcopyAIAI, 3)
.text
.align 4
.globl _bcopy__6arrays5BcopyAIAI
_bcopy__6arrays5BcopyAIAI:
          cmpl      $0, 4(%esp)
          je        L6
          movl      4(%esp), %ebx
          movl      4(%ebx), %ecx
          testl     %ecx, %ecx
          jg        L22
          ret
L22:
          xorl      %edx, %edx
          cmpl      $0, 8(%esp)
          je        L6
          movl      8(%esp), %eax
          movl      4(%eax), %esi
```

```
L7:
ANN_LOOP(INV = {
          (csubneq ebx 0),
          (csubneq eax 0),
          (csubb edx ecx),
          (of rm mem)},
         MODREG = (EDI,EDX,EFLAGS,FFLAGS,RM))
          cmpl      %esi, %edx
          jae       L13
          movl      8(%ebx, %edx, 4), %edi
          movl      %edi, 8(%eax, %edx, 4)
          incl      %edx
          cmpl      %ecx, %edx
          jl        L7
          ret
L13:
          call      __Jv_ThrowBadArrayIndex
ANN_UNREACHABLE
          nop
L6:
          call      __Jv_ThrowNullPointer
ANN_UNREACHABLE
          nop
```

Optimized target code

```
ANN_LOCALS(_bcopy__6arrays5BcopyAIAI, 3)
.text
.align 4
.globl _bcopy__6arrays5BcopyAIAI
_bcopy__6arrays5BcopyAIAI:
          cmpl      $0, 4(%esp)
          je        L6
          movl      4(%esp), %ebx
          movl      4(%ebx), %ecx
          testl     %ecx, %ecx
          jg        L22
          ret
L22:
          xorl      %edx, %edx
          cmpl      $0, 8(%esp)
          je        L6
          movl      8(%esp), %eax
          movl      4(%eax), %esi
```

VCGen requires annotations in order to simplify the process.

```
L7:
ANN_LOOP(INV = {
          (csubneq ebx 0),
          (csubneq eax 0),
          (csubb edx ecx),
          (of rm mem)},
         MODREG = (EDI,EDX,EFLAGS,FFLAGS,RM))
          cmpl      %esi, %edx
          jae       L13
          movl      8(%ebx, %edx, 4), %edi
          movl      %edi, 8(%eax, %edx, 4)
          incl      %edx
          cmpl      %ecx, %edx
          jl        L7
          ret
L13:
          call      __Jv_ThrowBadArrayIndex
ANN_UNREACHABLE
          nop
L6:
          call      __Jv_ThrowNullPointer
ANN_UNREACHABLE
          nop
```

Optimized loop body

```
L7:
ANN_LOOP(INV = {
       (csubneq ebx 0),
       (csubneq eax 0),
       (csubb edx ecx),
       (of rm mem)},
         MODREG = (EDI,EDX,EFLAGS,FFLAGS,RM))
       cmpl   %esi, %edx
       jae    L13
       movl   8(%ebx, %edx, 4), %edi
       movl   %edi, 8(%eax, %edx, 4)
       incl   %edx
       cmpl   %ecx, %edx
```

Essential facts about live variables, used by the compiler to eliminate bounds-checks in the loop body.

Loop invariants

One can see that the compiler “proves” facts such as

- $r \neq 0$
- $r < r'$ (unsigned)
- and a small number of others

The compiler deposits facts about the live variables in the loop

Symbolic evaluation

In contrast to the previous lecture, VCgen is actually performed via a forward scan

This slightly simplifies the handling of branches

The VCGen Process (1)

```
_bcopy__6arrays5BcopyAIAI:  A0 = (type src_1 (jarray jint))
                            A1 = (type dst_1 (jarray jint))
    cmpl  $0, src           A2 = (type rm_1 mem)
    je    L6                A3 = (csubneq src_1 0)
    movl  src, %ebx         ebx := src_1
    movl  4(%ebx), %ecx     ecx := (sel4 rm_1
    testl %ecx, %ecx                 (add src_1 4))
    jg    L22
    ret
L22:                        A4 = (csubgt (sel4 rm_1
                                   (add src_1 4)) 0)

    xorl  %edx, %edx        edx := 0
    cmpl  $0, dst
    je    L6                A5 = (csubneq dst_1 0)
    movl  dst, %eax         eax := dst_1
    movl  4(%eax), %esi     esi := (sel4 rm_1
L7: ANN_LOOP(INV = …                   (add dst_1 4))
```

The VCGen Process (2)

```
L7: ANN_LOOP(INV = {
     (csubneq ebx 0),          A3
     (csubneq eax 0),          A5
     (csubb edx ecx),          A6 = (csubb 0 (sel4 rm_1
     (of rm mem)},                    (add src_1 4)))
    MODREG =
     (EDI,                     edi := edi_1
      EDX,                     edx := edx_1
      EFLAGS,FFLAGS,RM))       rm := rm_2
    cmpl  %esi, %edx
    jae  L13                   A7 = (csubb edx_1 (sel4
                                      rm_2 (add dst_1 4))
    movl  8(%ebx,%edx,4), %edi !!Verify!! (saferd4
                                 (add src_1
                                   (add (imul edx_1 4) 8)))
    movl %edi, 8(%eax,%edx,4)
 …
```

The Checker (1)

The checker is asked to verify that

```
(saferd4 (add src_1 (add (imul edx_1 4) 8)))
```

under assumptions

```
A0 = (type src_1 (jarray jint))
A1 = (type dst_1 (jarray jint))
A2 = (type rm_1 mem)
A3 = (csubneq src_1 0)
A4 = (csubgt (sel4 rm_1 (add src_1 4)) 0)
A5 = (csubneq dst_1 0)
A6 = (csubb 0 (sel4 rm_1 (add src_1 4)))
A7 = (csubb edx_1 (sel4 rm_2 (add dst_1 4))
```

The checker looks in the PCC for a proof of this VC.

The Checker (2)

In addition to the assumptions, the proof may use axioms and proof rules defined by the host, such as

```
szint : pf (size jint 4)

rdArray4: {M:exp} {A:exp} {T:exp} {OFF:exp}
          pf (type A (jarray T)) ->
          pf (type M mem) ->
          pf (nonnull A) ->
          pf (size T 4) ->
          pf (arridx OFF 4 (sel4 M (add A 4))) ->
          pf (saferd4 (add A OFF)).
```

Checker (3)

A proof for

```
(saferd4 (add src_1 (add (imul edx_1 4) 8)))
```

in the Java specification looks like this (excerpt):

```
(rdArray4 A0 A2 (sub0chk A3) szint
   (aidxi 4 (below1 A7)))
```

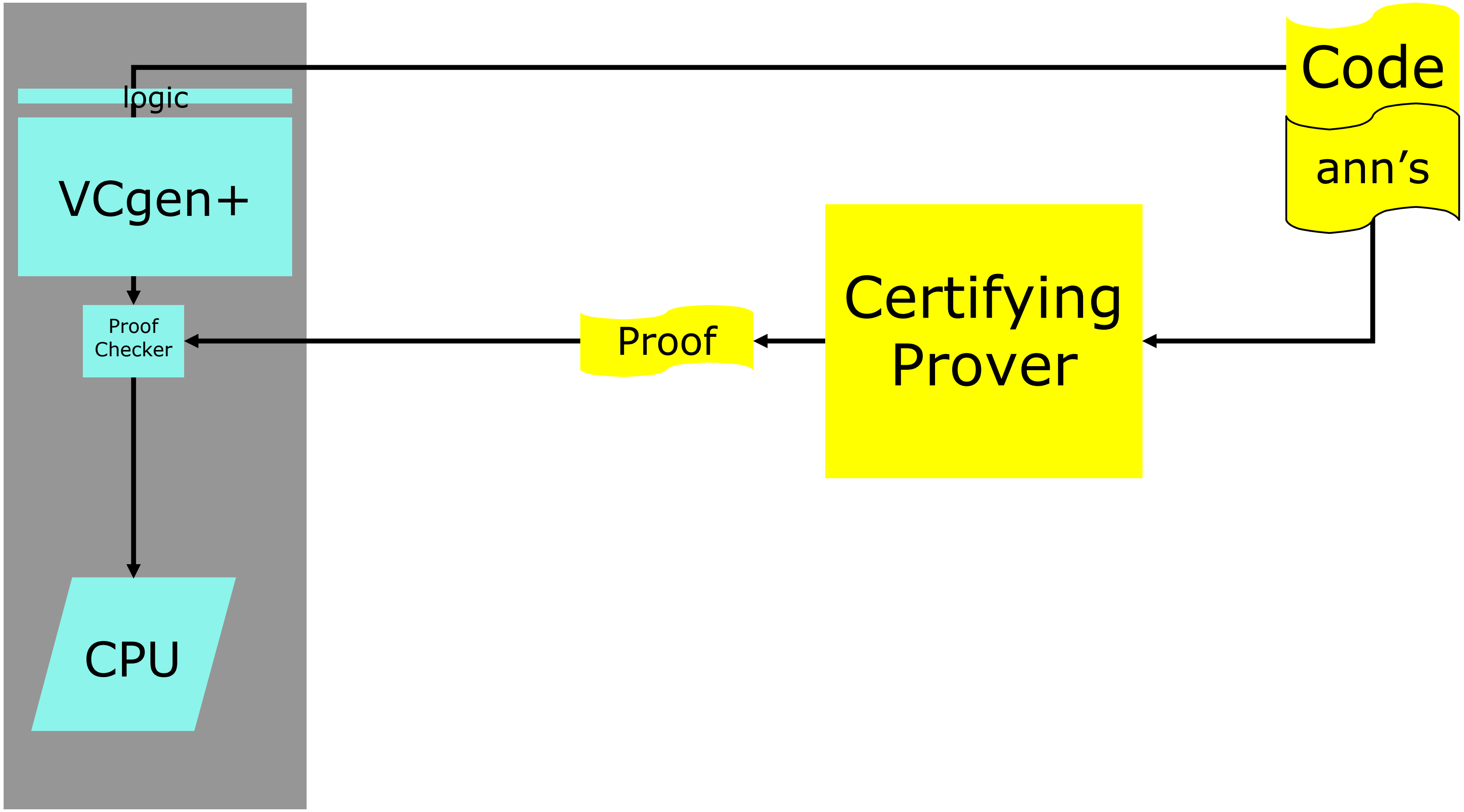
This proof can be easily validated via LF type checking.

Example: Proof excerpt
(LF representation)

```
ANN_PROOF(_6arrays6Bcopy1_MbcopyAIAI,
 %LF_(andi (impi [H_1 : pf (of _p22  (jarray jint))]
(andi (impi [H_2 : pf (of _p23  (jarray jint))]
(andi (impi [H_3 : pf (of _p21  mem)]
(andi (impi [H_4 : pf (ceq (sub _p23 0))]
truei)
(andi (impi [H_5 : pf (cneq (sub _p23 0))]
(andi (rd4 (arrLen H_2 (nullcsubne H_5)) szint)
(andi (nullcsubne H_5)
(andi H_3
(andi H_1
(andi (impi [H_10 : pf (nonnull _p23)]
(andi (impi [H_11 : pf (of _p64  mem)]
(andi (impi [H_12 : pf (of _p65  (jarray jint))]
(andi (impi [H_13 : pf (cnlt (sub _p49 (sel4 _p21 (add _p23 4))))]
(andi H_11
truei))
(andi (impi [H_15 : pf (clt (sub _p49 (sel4 _p21 (add _p23 4))))]
(andi (rd4 (arrLen H_2 H_10) szint)
(andi (impi [H_17 : pf (cnb (sub _p49 (sel4 _p64 (add _p23 4))))]
truei)
(andi (impi [H_18 : pf (cb (sub _p49 (sel4 _p64 (add _p23 4))))]
(andi (rd4 (arrElem H_2 H_11 H_10 szint (ultcsubb H_18)) szint)
(andi (impi [H_20 : pf (ceq (sub _p65 0))]
truei)
(andi (impi [H_21 : pf (cneq (sub _p65 0))]
(andi (rd4 (arrLen H_12 (nullcsubne H_21)) szint)
(andi (impi [H_23 : pf (cnb (sub _p49 (sel4 _p64 (add _p65 4))))]
truei)
(andi (impi [H_24 : pf (cb (sub _p49 (sel4 _p64 (add _p65 4))))]
(andi (wr4 (arrElem H_12 H_11 (nullcsubne H_21) szint (ultcsubb H_24)) szint
(jintany (sel4 _p64 (add _p23 (add (mul _p49 4) 8)))))
(andi H_10
(andi (ofamem 1)
(andi H_12
```

Improvements

Implementation, in reality



VCgen+ in SpecialJ

COFF (700 Loc)
- parsing
- relocation

ELF (600 Loc)
- …

MS PE (700 Loc)
- …

Core VCGen (12,300 Loc)
- Symbolic evaluation
- Register file management
- Control-flow support (jump, bcond, call, loop handling)
- Stack-frame management
- Generic obj. file support

x86 (3300 Loc)
- decoding
- calling convention
- special-register handling (FTOP)

DEC Alpha (1200 Loc)

ARM (1100 Loc)

M68k (2500 Loc)

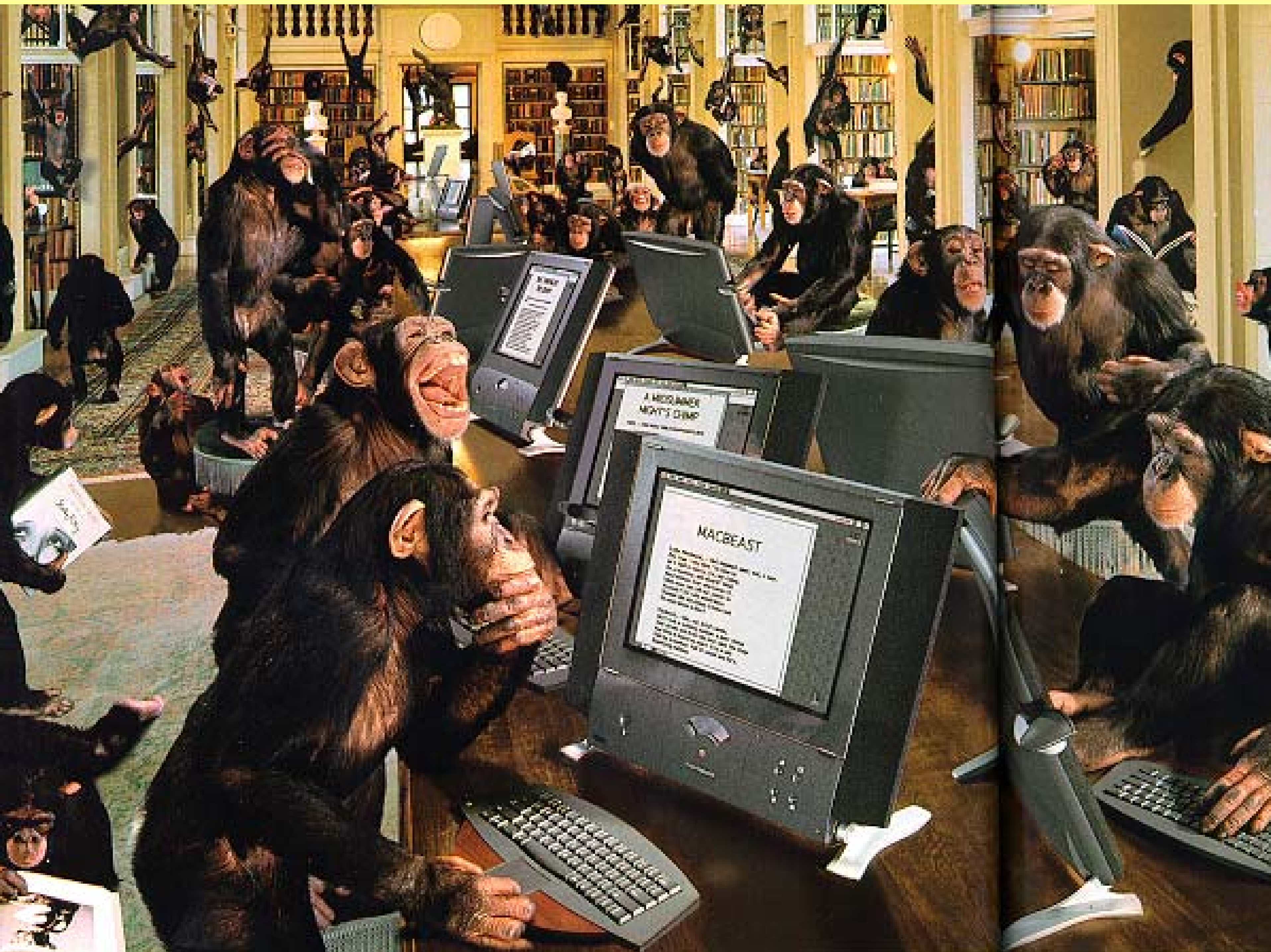
Java (3800 Loc)
- .class metadata parsing and checking
- exception handling
- annot. parsing and processing

Indirect call (270 Loc)

Indirect jump (350 Loc)

Total: (x86+Java) = 20,000 Loc

C code!


A MIDSUMMER NIGHT'S CHIMP
MACBEAST

The reality of scaling up

In SpecialJ, the proofs and annotations are OK, but the VCgen+ is

- complex, nontrivial C program
- machine-specific
- compiler-specific
- source-language specific
- safety-policy specific

トゥィンクルシルバー

A systems design principle

Separate *policy* from *mechanism*

One possible approach:

- devise some kind of *universal enforcement mechanism*

Typical elements of a system

Untrusted Elements

- Safety is not compromised if these fail.
- Examples:
  - Certifying compilers and provers

Trusted Elements

- To ensure safety, these must be right.
- Examples:
  - Verifier (type checker, VCgen, proof checker)
  - Runtime library
  - Hardware

The trouble with trust

Security:

- A trusted element might be wrong.
- It’s not clear how much we can do about this.
  - We can minimize our contribution, but must still trust the operating system.
  - Windows has more bugs than any certified code system.

The trouble with trust, cont’d

Extensibility:

- Everyone is stuck with the trusted elements.
  - They cannot be changed by developers.
  - If a trusted element is unsuitable to a developer, too bad.

Achieving extensibility

Main aim:

- Anyone should be able to target our system
- Want to support multiple developers, languages, applications.

But:

No single type or proof system is suitable for every purpose. (Not yet anyway!)

Thus:

Don’t trust the type/proof system.

Foundational Certified Code

In “Foundational” CC, we trust only:

1. A safety policy
   - Given in terms of the machine architecture.
2. A proof system
   - For showing compliance with the safety policy.
3. The non-verifier components (runtime library, hardware, etc.)

Foundational PCC

We can eliminate VCGen by using a *global invariant* on states, Inv(S)

Then, the proof must show:

- Inv(S_0)
- ΠS:State. Inv(S) $\rightarrow$ Inv(Step(S))
- ΠS:State. Inv(S) $\rightarrow$ SP(S)

In “Foundational PCC”, by Appel and Felty, we trust only the safety policy and the proofchecker, not the VCgen

Other “foundational” work

Hamid, Shao, et al. [’02] define the global invariant to be a syntactic well-formedness condition on machine states

Crary, et al. [’03] apply similar ideas in the development of TALT

Bernard and Lee [’02] use temporal logic specifications as a basis for a foundational PCC system

What is the right safety policy?

Whatever the host’s administrator wants it to be!

But in practice the question is not always easy to answer…

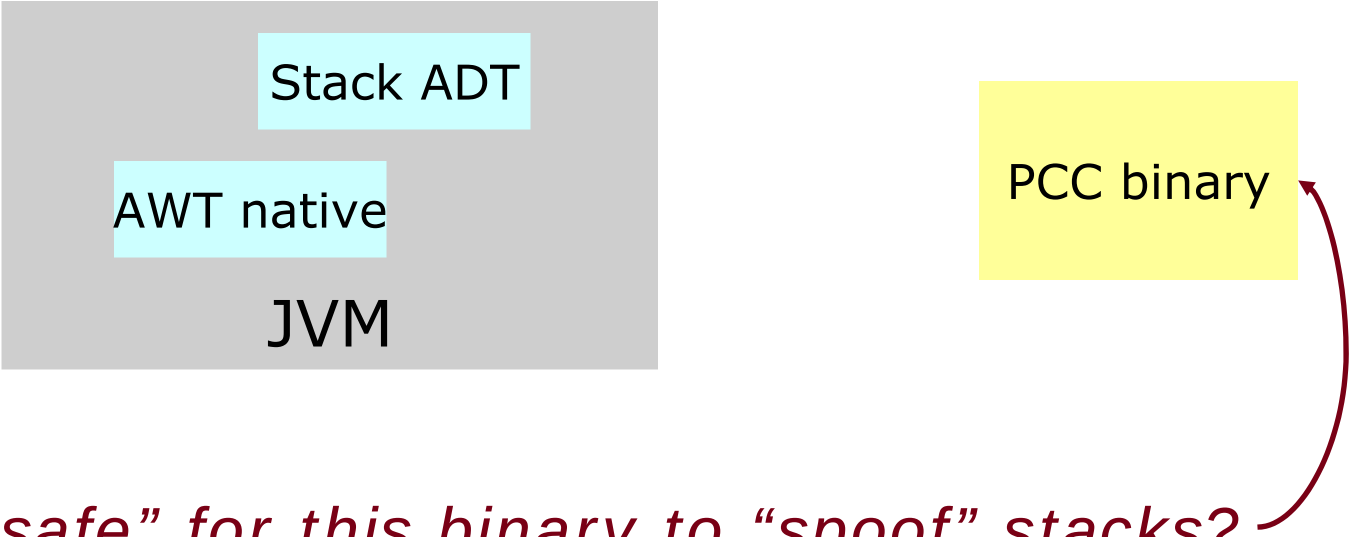
What is the right safety policy?

Some possibilities:

- Programs must be semantically equivalent to the source program [Pnueli, Rinard, …]
- Well-typed in a target language with a sound type system [Morrisett, Crary, …]
- Meets a logical specification (perhaps given in a Hoare logic) [Necula, Lee, …]

Safety in SpecialJ

The compiled output of SpecialJ is designed to link with the Java Virtual Machine



Is it “safe” for this binary to “spoof” stacks?

Proof rules
(excerpts)

3. Syntax and rules for the Java type system.

```
jint    : exp.
jfloat  : exp.
jarray  : exp -> exp.
jinstof : exp -> exp.

of     : exp -> exp -> pred.

faddf  : {E:exp} {E':exp}
           pf (of E jfloat) ->
           pf (of E' jfloat) ->
           pf (of (fadd E E') jfloat).

ext    : {E:exp} {C:exp} {D:exp}
           pf (jextends C D) ->
           pf (of E (jinstof C)) ->
           pf (of E (jinstof D)).
```

Flexibility in safety policies

Memory safety seems to be adequate for many applications

- But even this much is tricky to specify

Writing an LF signature + VCgen, or else rules for a type system, only “indirectly” specifies the safety policy

A language for safety policies

Linear-time 1st-order temporal logic [Manna/Pnueli 80]

- identify time with CPU clock

An attractive policy notation

- concise: $\Box(pc < 1000)$
- well-understood semantics
- can express variety of security policies
  - including type safety

Temporal logic PCC
[Bernard & Lee 02]

Encode safety policy (i.e., transition relation for safe execution) formally in temporal logic (following [Pnueli 77])

Prove directly that the program satisfies the safety policy

Encode the PCC certificate as a logic program from the combination of safety policy and proof

TL-PCC

Certificate is encoded as a *logic program* (in LF) that, when executed, generates a proof

- The certificate extracts its own VCs
- Certificate specializes the VCgen, logic, and annotations to the given program
- The fact that the certificate does its job correctly can be validated syntactically

Engineering tradeoffs

The certificates in foundational systems prove “more”, and hence there is likely to be greater overhead

Engineering tradeoffs in TL-PCC

Explicit security policies, easier to trust, change, and maintain

No VC generator, much less C code

No built-in flow analysis

But: Proof checking is much slower

Proof checking time

Current prototype in naïve in several ways, and should improve

Also represents one end of the spectrum.

- Is there a “sweet spot”?

Necula/Lee 96

Bernard/Lee 02

A current question

Since we use SpecialJ for our experiments, the certificates provide only type safety

But, in principle, can now enforce properties in temporal-logic

- How to generate the certificates?

Conclusions

PCC shows promise as a practical code certification technology

Several significant engineering hurdles remain, however

Lots of interesting future research directions

Thank you!