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
Today we continue by describing how to obtain the annotated programs via 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];
        }
    }
}

\[
\begin{align*}
\land & : \text{pred} \rightarrow \text{pred} \rightarrow \text{pred}. \\
\lor & : \text{pred} \rightarrow \text{pred} \rightarrow \text{pred}. \\
\Rightarrow & : \text{pred} \rightarrow \text{pred} \rightarrow \text{pred}. \\
\forall & : (\text{exp} \rightarrow \text{pred}) \rightarrow \text{pred}. \\
\text{pf} & : \text{pred} \rightarrow \text{type}. \\
\text{truei} & : \text{pf} \text{ true}. \\
\text{andi} & : \{P:\text{pred}\} \{Q:\text{pred}\} \text{ pf } P \rightarrow \text{pf } Q \rightarrow \text{pf } (\land P Q). \\
\text{andel} & : \{P:\text{pred}\} \{Q:\text{pred}\} \text{ pf } (\land P Q) \rightarrow \text{pf } P. \\
\text{ander} & : \{P:\text{pred}\} \{Q:\text{pred}\} \text{ pf } (\land P Q) \rightarrow \text{pf } Q.
\end{align*}
\]

...
2. Syntax and rules for arithmetic and equality.

\[
= : \text{exp} \to \text{exp} \to \text{pred}.
\]

\[
<> : \text{exp} \to \text{exp} \to \text{pred}.
\]

\[
eq_{\text{le}} : \{E:\text{exp}\} \{E':\text{exp}\} \ \text{pf} \ (\text{csubeq} \ E \ E') \to \text{pf} \ (\text{csuble} \ E \ E').
\]

\[
\text{moddist}+: \{E:\text{exp}\} \{E':\text{exp}\} \{D:\text{exp}\}
\text{pf} \ (= \ (\text{mod} (+ E E') D) \ (\text{mod} (+ (\text{mod} E D) E') D)).
\]

\[
\text{=}\text{sym} : \{E:\text{exp}\} \{E':\text{exp}\} \ \text{pf} \ (= E E') \to \text{pf} \ (= E' E).
\]

\[
<>\text{sym} : \{E:\text{exp}\} \{E':\text{exp}\} \ \text{pf} \ (<> E E') \to \text{pf} \ (<> E' E).
\]

\[
\text{=}\text{tr} : \{E:\text{exp}\} \{E':\text{exp}\} \{E'':\text{exp}\}
\text{pf} \ (= E E') \to \text{pf} \ (= E' E') \to \text{pf} \ (= E E'').
\]

“\text{csuble}” means ≤ in the x86 machine.
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.
3. Syntax and rules for the Java type system.

\[
\begin{align*}
\text{jint} & : \text{exp}. \\
\text{jfloat} & : \text{exp}. \\
\text{jarray} & : \text{exp} \rightarrow \text{exp}. \\
\text{jinstof} & : \text{exp} \rightarrow \text{exp}. \\
\text{of} & : \text{exp} \rightarrow \text{exp} \rightarrow \text{pred}. \\
\text{faddf} & : \{E: \text{exp}\} \{E': \text{exp}\}
\begin{align*}
& \text{pf} (\text{of} \ E \ \text{jfloat}) \rightarrow \\
& \text{pf} (\text{of} \ E' \ \text{jfloat}) \rightarrow \\
& \text{pf} (\text{of} (\text{fadd} \ E \ E') \ \text{jfloat}).
\end{align*} \\
\text{ext} & : \{E: \text{exp}\} \{C: \text{exp}\} \{D: \text{exp}\}
\begin{align*}
& \text{pf} (\text{jextends} \ C \ D) \rightarrow \\
& \text{pf} (\text{of} \ E (\text{jinstof} \ C)) \rightarrow \\
& \text{pf} (\text{of} \ E (\text{jinstof} \ D)).
\end{align*}
\end{align*}
\]
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.

\[ \text{aidxi} : \{I:\text{exp}\} \{\text{LEN:exp}\} \{\text{SIZE:exp}\} \]
\[ \quad \text{pf (below I LEN)} \rightarrow \]
\[ \quad \text{pf (arridx (add (imul I SIZE) 8) SIZE LEN)}. \]

\[ \text{wrArray4: \{M:\text{exp}\} \{A:\text{exp}\} \{T:\text{exp}\} \{OFF:\text{exp}\} \{E:\text{exp}\}} \]
\[ \quad \text{pf (of A \{jarray T\}) \rightarrow} \]
\[ \quad \text{pf (of M mem) \rightarrow} \]
\[ \quad \text{pf (nonnull A) \rightarrow} \]
\[ \quad \text{pf (size T 4) \rightarrow} \]
\[ \quad \text{pf (arridx OFF 4 \{sel4 M (add A 4)\}) \rightarrow} \]
\[ \quad \text{pf (of E T) \rightarrow} \]
\[ \quad \text{pf (safewr4 (add A OFF) E)}. \]

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

\[
\begin{align*}
nlt0_0 & : \text{pf\ (csubnlt\ 0\ 0)}. \\
nlt1_0 & : \text{pf\ (csubnlt\ 1\ 0)}. \\
nlt2_0 & : \text{pf\ (csubnlt\ 2\ 0)}. \\
nlt3_0 & : \text{pf\ (csubnlt\ 3\ 0)}. \\
nlt4_0 & : \text{pf\ (csubnlt\ 4\ 0)}. \\
\end{align*}
\]

Inevitably, “unclean” things are sometimes put into the specification…
How do we know that it is right?
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,LOCATION)LOC4,LOCATION3)
  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__6arrays5BcopyAI, 3)
.text
.align 4
.globl _bcopy__6arrays5BcopyAI
_bcopy__6arrays5BcopyAI:
    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,EFLA,G,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
    _bcopy__6arrays5BcopyAIAI:
    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
L6:
    call __Jv_ThrowNullPointer
    _bcopy__6arrays5BcopyAIAI:
    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
    _bcopy__6arrays5BcopyAIAI:
    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
    _bcopy__6arrays5BcopyAIAI:
    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

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.
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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpl $0, src</td>
<td>Compare $0 with src</td>
</tr>
<tr>
<td>je L6</td>
<td>Jump if equal to L6</td>
</tr>
<tr>
<td>movl src, %ebx</td>
<td>Move src to %ebx</td>
</tr>
<tr>
<td>movl 4(%ebx), %ecx</td>
<td>Move 4 times the value of ebx to %ecx</td>
</tr>
<tr>
<td>testl %ecx, %ecx</td>
<td>Test %ecx for equality</td>
</tr>
<tr>
<td>jg L22</td>
<td>Jump if greater than L22</td>
</tr>
<tr>
<td>ret</td>
<td>Return</td>
</tr>
</tbody>
</table>

L22:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xorl %edx, %edx</td>
<td>XOR %edx with itself</td>
</tr>
<tr>
<td>cmpl $0, dst</td>
<td>Compare $0 with dst</td>
</tr>
<tr>
<td>je L6</td>
<td>Jump if equal to L6</td>
</tr>
<tr>
<td>movl dst, %eax</td>
<td>Move dst to %eax</td>
</tr>
<tr>
<td>movl 4(%eax), %esi</td>
<td>Move 4 times the value of eax to %esi</td>
</tr>
</tbody>
</table>

L7: ANN_LOOP(INV = ...
The VCGen Process (2)

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
!!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

\[(\text{saferd4} \ (\text{add src}_1 \ (\text{add} \ (\text{imul edx}_1 \ 4) \ 8)))\]

under assumptions

\[A0 = (\text{type src}_1 \ (\text{jarray jint}))\]
\[A1 = (\text{type dst}_1 \ (\text{jarray jint}))\]
\[A2 = (\text{type rm}_1 \ \text{mem})\]
\[A3 = (\text{csubneq src}_1 \ 0)\]
\[A4 = (\text{csubgt} \ (\text{sel4 rm}_1 \ (\text{add src}_1 \ 4)) \ 0)\]
\[A5 = (\text{csubneq dst}_1 \ 0)\]
\[A6 = (\text{csubb} \ 0 \ (\text{sel4 rm}_1 \ (\text{add src}_1 \ 4)))\]
\[A7 = (\text{csubb edx}_1 \ (\text{sel4 rm}_2 \ (\text{add dst}_1 \ 4)))\]

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

\[
\text{szint} : \text{pf} \ (\text{size \ jint \ 4})
\]

\[
\text{rdArray4} : \ \{M:exp\} \ \{A:exp\} \ \{T:exp\} \ \{OFF:exp\}
\]
\[
\text{pf} \ (\text{type} \ A \ (\text{jarray} \ T)) \ \rightarrow
\]
\[
\text{pf} \ (\text{type} \ M \ \text{mem}) \ \rightarrow
\]
\[
\text{pf} \ (\text{nonnull} \ A) \ \rightarrow
\]
\[
\text{pf} \ (\text{size} \ T \ 4) \ \rightarrow
\]
\[
\text{pf} \ (\text{arridx} \ OFF \ 4 \ (\text{sel4} \ M \ (\text{add} \ A \ 4))) \ \rightarrow
\]
\[
\text{pf} \ (\text{saferd4} \ (\text{add} \ A \ \text{OFF})).
\]
A proof for

\( (\text{saferd4} (\text{add} \ src_1 (\text{add} \ (\text{imul} \ edx_1 4) \ 8))) \)

in the Java specification looks like this (excerpt):

\( (\text{rdArray4} A0 A2 (\text{sub0chk} A3) \text{ szint} \ \\
\quad (\text{aidxi} 4 (\text{below1} A7))) \)

This proof can be easily validated via LF type checking.
Example: Proof excerpt (LF representation)

```
ANN_PROOF(_6arrays6Bcopy1_MbcopyAI, 
  %LF_(andi (imi [H_1 : pf (of _p22 (jarray jint))]
  (andi (imi [H_2 : pf (of _p23 (jarray jint))]
  (andi (imi [H_3 : pf (of _p21 mem)])
  (andi (imi [H_4 : pf (ceq (sub _p23 0))])
  truei)
  (andi (imi [H_5 : pf (cneq (sub _p23 0))])
  (andi (rd4 (arrLen H_2 (nullcsblne H_5)) szint)
  (andi (nullcsblne H_5)
  (andi H_3
  (andi H_1
  (andi (imi [H_10 : pf (nonnull _p23)])
  (andi (imi [H_11 : pf (of _p64 mem)])
  (andi (imi [H_12 : pf (of _p65 (jarray jint))])
  (andi (imi [H_13 : pf (cneq (sub _p49 (sel4 _p21 (add _p23 4))))])
  (andi H_11 
  truei))
  (andi (imi [H_15 : pf (clt (sub _p49 (sel4 _p21 (add _p23 4))))])
  (andi (rd4 (arrLen H_2 H_10) szint)
  (andi (imi [H_17 : pf (cb (sub _p49 (sel4 _p64 (add _p23 4))))])
  truei)
  (andi (imi [H_18 : pf (cneq (sub _p49 (sel4 _p64 (add _p23 4))))])
  (andi (rd4 (arrElem H_2 H_11 H_10 szint (ultcsblne H_18)) szint)
  (andi (imi [H_20 : pf (ceq (sub _p65 0)])
  truei)
  (andi (imi [H_21 : pf (cneq (sub _p65 0)])
  (andi (rd4 (arrLen H_12 (nullcsblne H_21)) szint)
  (andi (imi [H_23 : pf (cb (sub _p49 (sel4 _p64 (add _p64 4))))])
  truei)
  (andi (imi [H_24 : pf (cb (sub _p49 (sel4 _p64 (add _p65 4))))]
  (andi (wr4 (arrElem H_12 H_11 (nullcsblne H_21) szint (ultcsblne 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+ → Proof Checker → CPU → Certifying Prover → Proof → Code

Logic

ann’s
VCgen+ in SpecialJ

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!

COFF (700 Loc)
- parsing
- relocation

ELF (600 Loc)
- ...

MS PE (700 Loc)
- ...

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!
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
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.
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.)
We can eliminate VCGen by using a global invariant on states, $\text{Inv}(S)$.

Then, the proof must show:

- $\text{Inv}(S_0)$
- $\forall S:\text{State}. \text{Inv}(S) \rightarrow \text{Inv}(\text{Step}(S))$
- $\forall S:\text{State}. \text{Inv}(S) \rightarrow \text{SP}(S)$

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.
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.

\[
\begin{align*}
jint & : \text{exp}. \\
jfloat & : \text{exp}. \\
jarray & : \text{exp} \rightarrow \text{exp}. \\
jinstof & : \text{exp} \rightarrow \text{exp}. \\
of & : \text{exp} \rightarrow \text{exp} \rightarrow \text{pred}. \\
faddf & : \{E:\text{exp}\} \{E':\text{exp}\} \\
& \quad \text{pf} (\text{of } E \ jfloat) \rightarrow \\
& \quad \text{pf} (\text{of } E' \ jfloat) \rightarrow \\
& \quad \text{pf} (\text{of } (fadd E E') \ jfloat). \\
ext & : \{E:\text{exp}\} \{C:\text{exp}\} \{D:\text{exp}\} \\
& \quad \text{pf} (\text{jextends } C \ D) \rightarrow \\
& \quad \text{pf} (\text{of } E \ (jinstof C)) \rightarrow \\
& \quad \text{pf} (\text{of } E \ (jinstof D)).
\end{align*}
\]
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 1\textsuperscript{st}-order temporal logic
[Manna/Pnueli 80]

- identify time with CPU clock

An attractive policy notation

- concise: \(\square (pc < 1000)\)
- 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
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
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!