Lectures on Proof-Carrying Code

Peter Lee
Carnegie Mellon University

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

```java
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)


- \( \land \) : \( \text{pred} \rightarrow \text{pred} \rightarrow \text{pred} \)
- \( \lor \) : \( \text{pred} \rightarrow \text{pred} \rightarrow \text{pred} \)
- \( \rightarrow \) : \( \text{pred} \rightarrow \text{pred} \rightarrow \text{pred} \)
- \( \forall \) : \( \text{exp} \rightarrow \text{pred} \rightarrow \text{pred} \)
- \( \text{pf} \) : \( \text{pred} \rightarrow \text{type} \)

- \( \text{true} \) : \( \text{pf} \rightarrow \text{true} \)
- \( \text{and} \) : \( \text{pred} \rightarrow \text{pred} \rightarrow \text{pred} \)
- \( \text{and} \) : \( \text{pred} \rightarrow \text{pred} \rightarrow \text{pred} \)
- \( \text{and} \) : \( \text{pred} \rightarrow \text{pred} \rightarrow \text{pred} \)

- \( \text{and} \) : \( \text{pred} \rightarrow \text{pred} \rightarrow \text{pred} \)
- \( \text{and} \) : \( \text{pred} \rightarrow \text{pred} \rightarrow \text{pred} \)
- \( \text{and} \) : \( \text{pred} \rightarrow \text{pred} \rightarrow \text{pred} \)
- \( \text{and} \) : \( \text{pred} \rightarrow \text{pred} \rightarrow \text{pred} \)

```plaintext
Syntax of predicates.

Type of valid proofs, indexed by predicate.

Inference rules.
```
Proof rules (excerpts)

2. Syntax and rules for arithmetic and equality.

```
* : 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)).
eq_sym : {E:exp} {E':exp} pf (= E E') -> pf (= E' E).
<>sym : {E:exp} {E':exp} pf (<> E E') -> pf (<> E' E).
eq_tr : {E:exp} {E':exp} {E'':exp }
    pf (= E E') -> pf (= E' E'') -> pf (= E E'').
```

Notes on x86 arithmetic:

```
"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.

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)

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)).
```

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).
 wskArray4: {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.

\[
\begin{align*}
\text{nl} & 0.0 : \text{pf (csubnlt 0 0)}. \\
\text{nl} & 1.0 : \text{pf (csubnlt 1 0)}. \\
\text{nl} & 2.0 : \text{pf (csubnlt 2 0)}. \\
\text{nl} & 3.0 : \text{pf (csubnlt 3 0)}. \\
\text{nl} & 4.0 : \text{pf (csubnlt 4 0)}. \\
\end{align*}
\]

Inevitably, "unclean" things are sometimes put into the specification...

How do we know that it is right?

Back to our example

```java
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

Bounds check on src.

Bounds check on dst.

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.

Note: L24 raises the ArrayIndex exception.
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
L13:
  call __Jv_ThrowBadArrayIndex
  ANN_UNREACHABLE
  nop
L6:
  call __Jv_ThrowNullPointer
  ANN_UNREACHABLE
  nop
```

VCGen requires annotations in order to simplify the process.

Optimized loop body

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

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)

```c
_bcopy__6arrays5BcopyAIAI:
    cmpl $0, src
    je L6
    movl src, %ebx
    movl 4(%ebx), %ecx
    testl %ecx, %ecx
    jg L22
    ret
L22:
```

The VCGen Process (2)

```c
L7: ANN_LOOP (INV = {
    (csubneq ebx 0),
    (csubgt (sel4 rm_1 (add src_1 4)) 0),
    (csubb edx_1 (sel4 rm_2 (add dst_1 4))
})
```

The Checker (1)

The checker is asked to verify that

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

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 4 (nullcsubne H_5))

Example: Proof excerpt (LF representation)

(A proof for

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

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.

Checker (3)
Improvements

VCgen+ in SpecialJ

Core VCgen (12,300 Loc)
  - Symbolic evaluation
  - Register file management
  - Control-flow support
    (jump, branch, 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 (3900 Loc)
  - Class metadata
  - Exception handling
  - Garbage collection

Indirect call (270 Loc)

Indirect jump (130 Loc)

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

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

Implementation, in reality
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(S₀)
- IIS:State. Inv(S) \land Inv(Step(S))
- IIS:State. Inv(S) \land 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...

Safety in SpecialJ

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

Is it "safe" for this binary to "spoo"f 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 => prod.
faddf : \{E:exp\} \{E’:exp\}
       pf (of E jfloat) =>
       pf (of E’ jfloat) =>
       pf (of (fadd E E’) jfloat).
exit : \{E:exp\} \{C:exp\} \{D:exp\}
      pf \{jinstof 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: □(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"?

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!