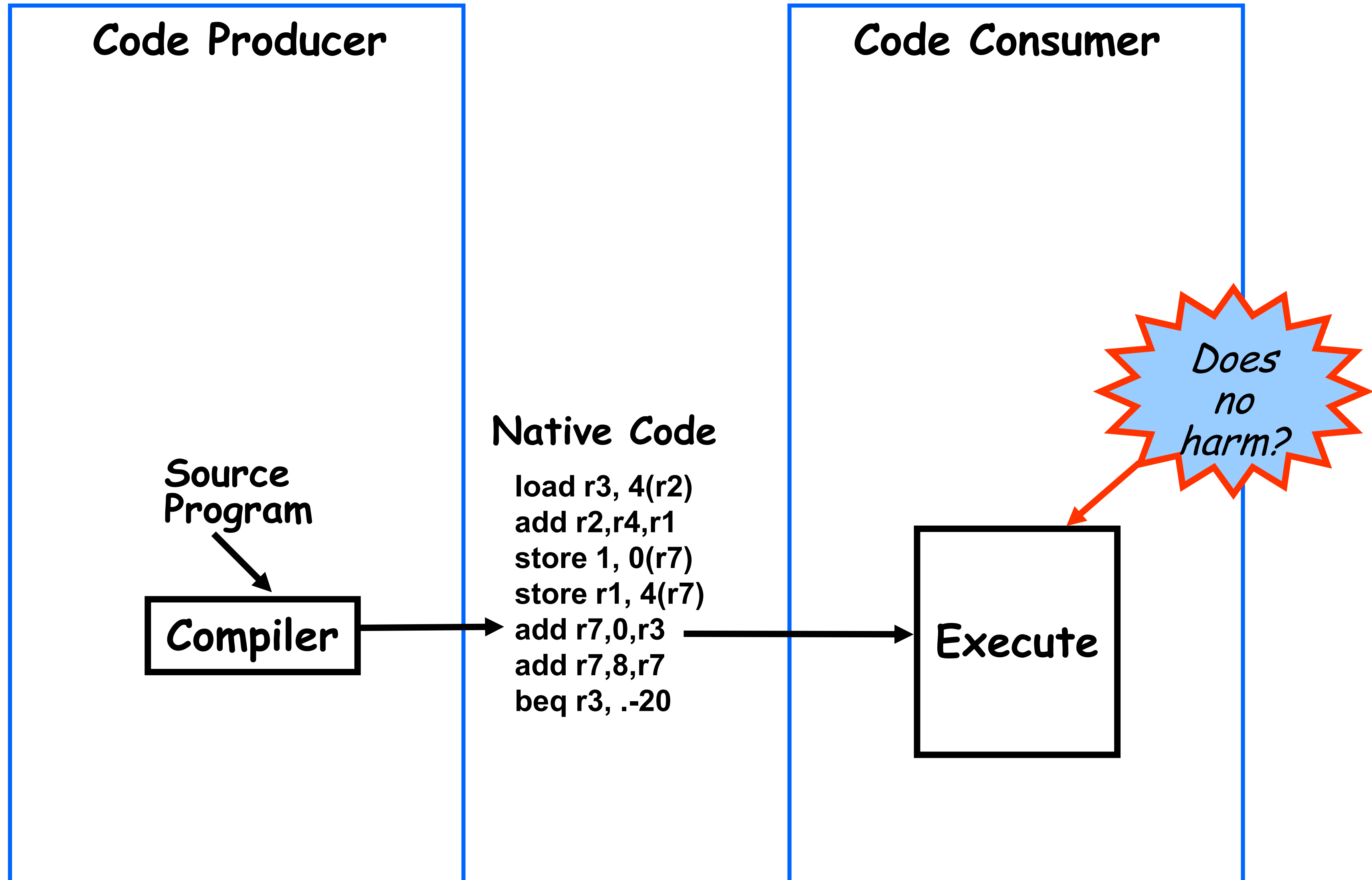
An Introduction to Proof-Carrying Code

Amy Felty

CSI 5110

The Problem: Code Safety



References

- Proof-Carrying Code
  - raw.cs.berkeley.edu/pcc.html
  - George Necula & Peter Lee, Proof-Carrying Code, Technical Report CMU-CS-96-165, 1996. Available from course web page.
  - George Necula, Proof-Carrying Code, Symposium on Principles of Programming Languages (POPL), 1997.
  - George Necula & Peter Lee, The Design and Implementation of a Certifying Compiler, PLDI 1998.
- Other References – See web page for course project.

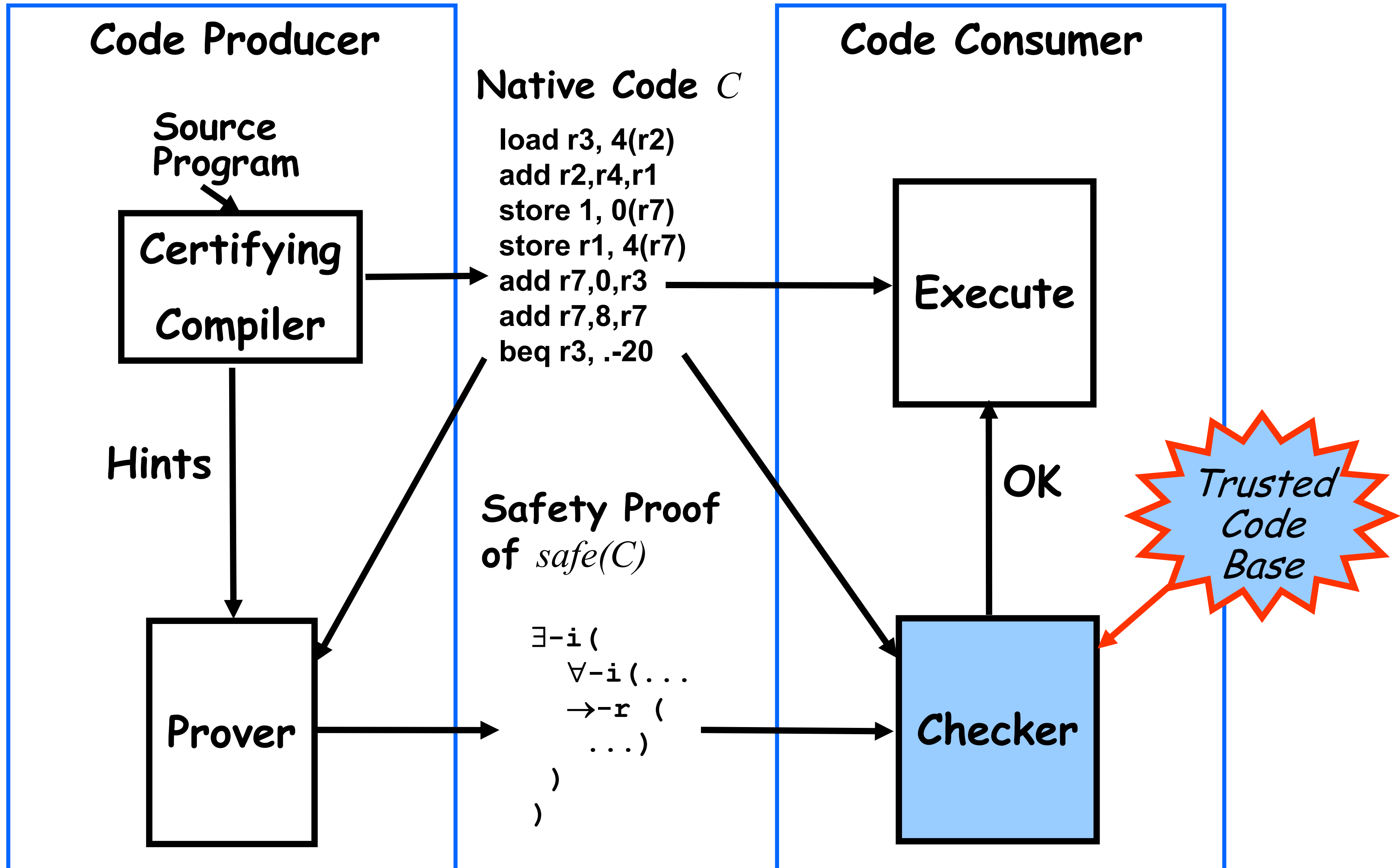
Proof-Carrying Code

Code Producer

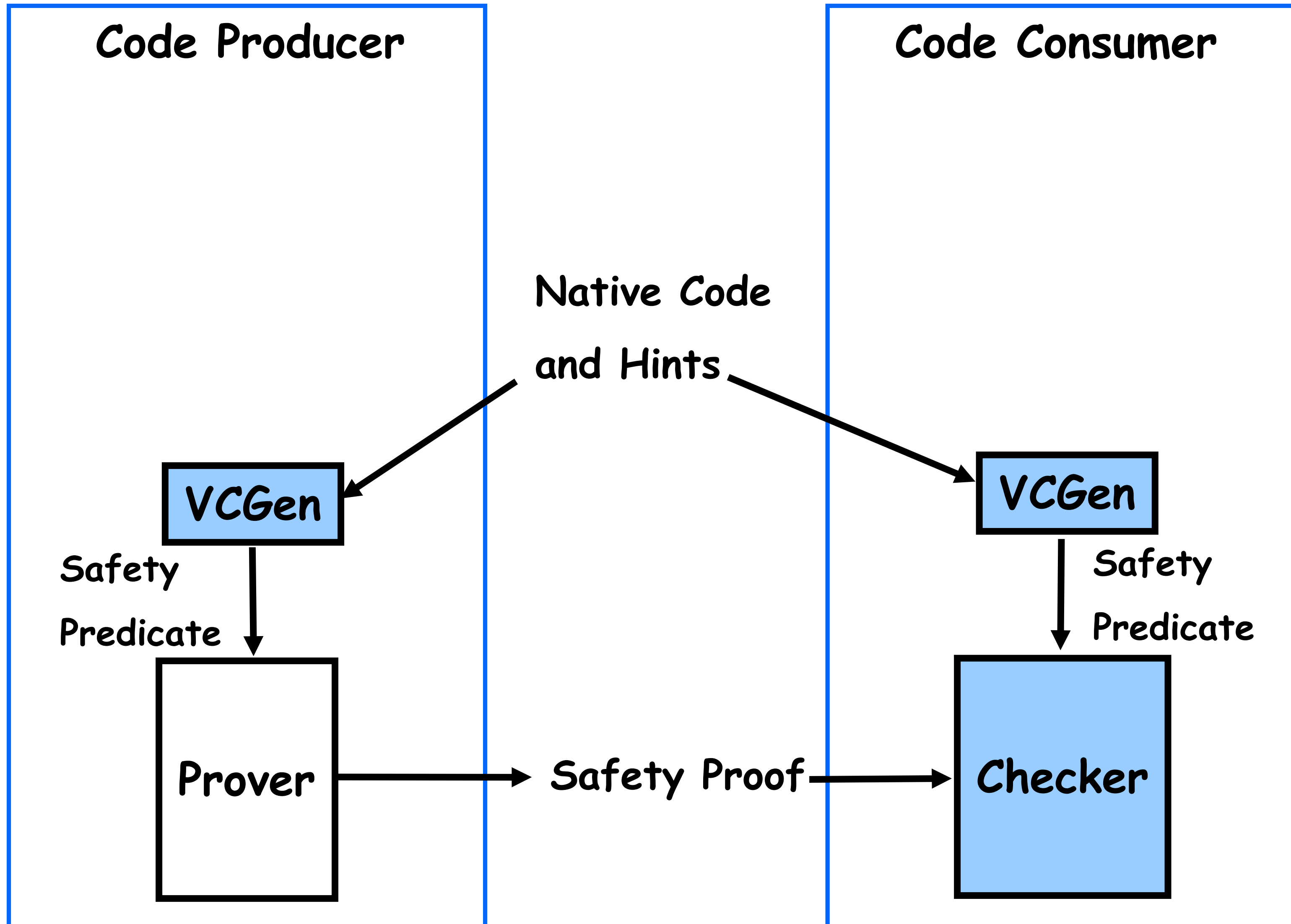
- Implements a program and compiles it to native machine code *C*.
- The verification condition *safe(C)* is sent to a prover which proves it (automatically) and outputs a proof *P*.
- The compiler also sends hints to the prover.

- The code producer communicates the *code* and *proof* to the code consumer.
- Code Consumer
  - Checks that *P* is a proof of *safe(C)*.
  - If successful, executes *C* as needed.
- Safety Policy
  - Set ahead of time by the code consumer.
  - Defined by a set of inference rules.

Proof-Carrying Code



A Closer Look at the Prover and Checker



What can go wrong?

- The proof gets corrupted during transfer from producer to consumer.
  - The proof is unlikely to check.
  - If it does, the new proof is an alternate proof of safety.
- The code gets corrupted during transfer from producer to consumer.
  - The proof is unlikely to check.
  - If it does, the new code is still safe (though it may not do what it was intended to do).

Advantages of Proof-Carrying Code

- Trusted Code Base is quite small; includes only the checker.
- No need to trust compiler or prover.
- The safety policy (meaning of *safe*) can be general and flexible.
  - Can use types, dataflow, induction, or any other provable property.
- Automated proof is possible for a large class of properties.
  - Safety properties of interest are relatively simple.
  - Hints from the compiler provide help.

Safety

- *Type safety and memory safety*
- Prohibit accessing private data
- Prevent overwriting important data
- Prevent accessing unauthorized resources
- Avoid consuming too many resources

Outline

- An Example Program
- Certifying Compiler
- Verification Condition Generator (VCGen)
- Checker
- Prover
- Executing the Program

An ML Program to Add the Values in a List

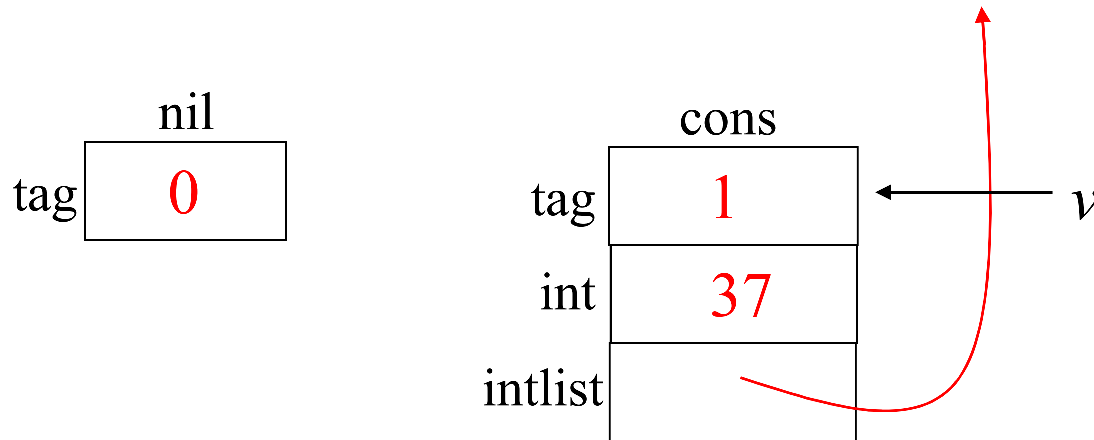
```
datatype intlist =
    nil of ()   |   cons of int * intlist;

fun addnums nil = 0
  | addnums (cons(n,ns)) = n + (addnums ns);
```

Outline: Certifying Compiler

- Layout of datatypes in memory
- Instruction set
- Formulas used to express hints and safety policies (propositional logic)
- The abstract machine
  - Important for proving correctness of the PCC technique
- Example program compiled to machine code (with hints)

Compiler Layout of Data Structures

```
datatype intlist =
    nil of ()   |   cons of int * intlist
```



Example Instruction Set

- ADD r_d := r_{s1} + r_{s2}
- ADDC r_d := r_s + c
- LD r_d := m(r_s + c)
- ST m(r_d + c) := r_s
- BEQ (r_{s1} = r_{s2}) c
- BGT (r_{s1} > r_{s2}) c
- RET
- INV *p*

Registers, Memory, and Expressions

- Register bank:

  $r ::== R \mid upd(r,n,e)$

- Memory:

  $m ::== M \mid upd(m,e_1, e_2)$

- Expressions:

  $e ::== n \mid e_1 + e_2 \mid r_n \mid pc \mid ret \mid m(e)$

Formulas of Safety Policy

- Types:

  $\tau ::== int \mid intlist \mid ...$

- Atomic Formulas:

  $A ::== e :_m \tau \mid e_1 = e_2 \mid e_1 < e_2 \mid readable(e) \mid writable(e)$

- Formulas:

  $P ::== A \mid \bot \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \rightarrow P_2 \mid \neg P$

The Abstract Machine: One Computation Step

$(r,m) \mapsto (r',m')$

if (r,m) evaluates to (r',m') by executing one instruction.

Let $r''=upd(r,pc,r_{pc}+1)$.

The Abstract Machine (1)

(r,m) evaluates to:

Instruction	*r'*	*m'*
ADD r_d := r_{s1} + r_{s2}	$upd(r'', d, r_{s1} + r_{s2})$	*m*
ADDC r_d := r_s + c	$upd(r'', d, r_s + c)$	*m*
LD r_d := m(r_s + c) *and* $readable(r_s + c)$	$upd(r'', d, m(r_s + c))$	*m*
ST m(r_d + c) := r_s *and* $writable(r_d + c)$	*r''*	$upd(m, r_d + c, r_s)$
RET	$upd(r, pc, ret)$	*m*
INV p	*r''*	*m*

The Abstract Machine (2)

(r,m) evaluates to:

Instruction	*r'*	*m'*
BEQ $(r_{s1} = r_{s2})$ c and $(r_{s1} = r_{s2})$	$upd(r, pc, r_{pc} + c)$	m
BEQ $(r_{s1} = r_{s2})$ c and $(r_{s1} \neq r_{s2})$	r''	m
BGT $(r_{s1} > r_{s2})$ c and $(r_{s1} > r_{s2})$	$upd(r, pc, r_{pc} + c)$	m
BGT $(r_{s1} > r_{s2})$ c and $(r_{s1} \leq r_{s2})$	r''	m

Compiled Program with Hints

Precondition: $r_0 :_m intlist \wedge r_3 = 0$

```
    ADD  r1 := r3 + r3          %initialize total to 0
    INV  r0 :m intlist ∧ r1 :m int ∧ r3 = 0
L1  LD   r5 := m(r0 + 0)        %r5 gets list tag
    BEQ  (r5 = r3) L2           %jump if list tag is 0
    LD   r2 := m(r0 + 1)        %load next int in r2
    LD   r0 := m(r0 + 2)        %r0 gets pointer to rest
    ADD  r1 := r1 + r2          %add next int to total
    BEQ  (r3 = r3) L1           %jump back
    INV  r1 :m int
L2  ADDC r0 := r1 + 0           %put total in r0
    RET
```

Outline: Verification Condition Generator

- The Verification Condition
- Motivating the VCGen Algorithm
  - Hoare Logic
  - Hoare Logic for Machine Instructions
- The VCGen algorithm
- Correctness of VCGen
- Example safety predicate generated by the algorithm

The Verification Condition

- A predicate in propositional logic with the property that its validity with respect to the inference rules of the safety policy is a sufficient condition for ensuring compliance with the safety policy.
- Includes:
  - proof that loop invariant holds when loop first entered
  - proof that invariant is preserved around the loop
  - proof that postcondition follows from invariant

Hoare Logic for Program Verification (Ch. 4)

$(|Q[e/x]|)\ x{:=}e\ (|Q|)$ e.g., $(|x=0|)$ `y:=0` $(|x=y|)$

$$\frac{(|P \wedge B|)\ S_1 (|Q|) \qquad (|P \wedge \neg B|)\ S_2 (|Q|)}{(|P|)\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2 (|Q|)}$$

$$\frac{(|P|) S_1 (|P'|) \qquad (|P'|)\ S_2 (|Q|)}{(|P|)\ S_1 ; S_2 (|Q|)}$$

$$\frac{P \rightarrow P' \quad (|P'|)\ S (|Q'|) \quad Q' \rightarrow Q}{(|P|)\ S (|Q|)}$$

Hoare Logic for Machine Instructions (1)

$$\frac{Pre \rightarrow Q_1 \quad (|Q_1|)S_1(|Q_2|) \quad (|Q_2|)S_2(|Q_3|) \ \ldots \ (|Q_n|)S_n(|Q_{n+1}|) \quad Q_{n+1} \rightarrow Post}{(|Pre|)S_1;S_2;\ldots;S_n(|Post|)}$$

$$\frac{P \rightarrow P' \quad (|P'|)\ S\ (|Q'|) \quad Q' \rightarrow Q}{(|P\ |)\ S\ (|Q\ |)}$$

$(|Q[(r_{s1}+r_{s2})/r_d]\ |)$ **`ADD`** $\mathtt{r_d := r_{s1} + r_{s2}}$ $(|Q|)$

e.g., $(|r_1+r_2=0|)$ **`ADD`** $\mathtt{r_3 := r_1 + r_2}$ $(|\ r_3=0|)$

$(|Q[(r_s+c)/r_d]\ |)$ **`ADDC`** $\mathtt{r_d := r_s + c}$ $(|Q|)$

Hoare Logic for Machine Instructions (2)

$(| \; Q[m(r_s+c)/r_d] \wedge readable(r_s+c) |)$ **LD r_d:=m(r_s+c)** $(|Q|)$

$(|Q[upd(m,r_d+c,r_s)/m] \wedge writable(r_d+c)|)$ **ST m(r_d+c):=r_s** $(|Q|)$

$(|(r_{s1} = r_{s2} \rightarrow Q_c) \wedge (\neg(r_{s1} = r_{s2}) \rightarrow Q)|)$ **BEQ (r_{s1}=r_{s2}) c** $(|Q|)$

$(|(r_{s1} > r_{s2} \rightarrow Q_c) \wedge (\neg(r_{s1} > r_{s2}) \rightarrow Q)|)$ **BGT (r_{s1}>r_{s2}) c** $(|Q|)$

$(|Q|)$ **RET** $(|Q|)$

Definition of Verification Condition Generator

- Let Π be the list of instructions output by the certifying compiler. Let Π_i be the instruction at position i in Π.
- Note: VC_{i+1} is needed to compute VC_i.

.

$$VC_i = \begin{cases} VC_{i+1}[(r_{s1}+r_{s2})/r_d] & \text{if } \Pi_i \text{ is } \texttt{ADD } \texttt{r}_\texttt{d}\texttt{:=r}_\texttt{s1}\texttt{+r}_\texttt{s2} \\ VC_{i+1}[(r_s+c)/r_d] & \text{if } \Pi_i \text{ is } \texttt{ADDC } \texttt{r}_\texttt{d}\texttt{:=r}_\texttt{s}\texttt{+c} \\ VC_{i+1}[m(r_s+c)/r_d] \wedge readable(r_s+c) & \\ & \text{if } \Pi_i \text{ is } \texttt{LD } \texttt{r}_\texttt{d}\texttt{:=m(r}_\texttt{s}\texttt{+c)} \\ VC_{i+1}[upd(m,r_d+c,r_s)/m] \wedge writable(r_d+c) & \\ & \text{if } \Pi_i \text{ is } \texttt{ST m(r}_\texttt{d}\texttt{+c):=r}_\texttt{s} \end{cases}$$

Definition of VCG (continued)

$$VC_i = \begin{cases} (r_{s1} = r_{s2} \rightarrow VC_{i+c-1}) \wedge (\neg(r_{s1} = r_{s2}) \rightarrow VC_{i+1}) & \text{if } \Pi_i \text{ is } \texttt{BEQ (r}_{\texttt{s1}}\texttt{=r}_{\texttt{s2}}\texttt{) c} \\ (r_{s1} > r_{s2} \rightarrow VC_{i+c-1}) \wedge (\neg(r_{s1} > r_{s2}) \rightarrow VC_{i+1}) & \text{if } \Pi_i \text{ is } \texttt{BGT (r}_{\texttt{s1}}\texttt{>r}_{\texttt{s2}}\texttt{) c} \\ post & \text{if } \Pi_i \text{ is } \texttt{RET} \\ p & \text{if } \Pi_i \text{ is } \texttt{INV } p \end{cases}$$

- *post* is the postcondition.
- Every jump point must be proceeded by an **INV** statement.

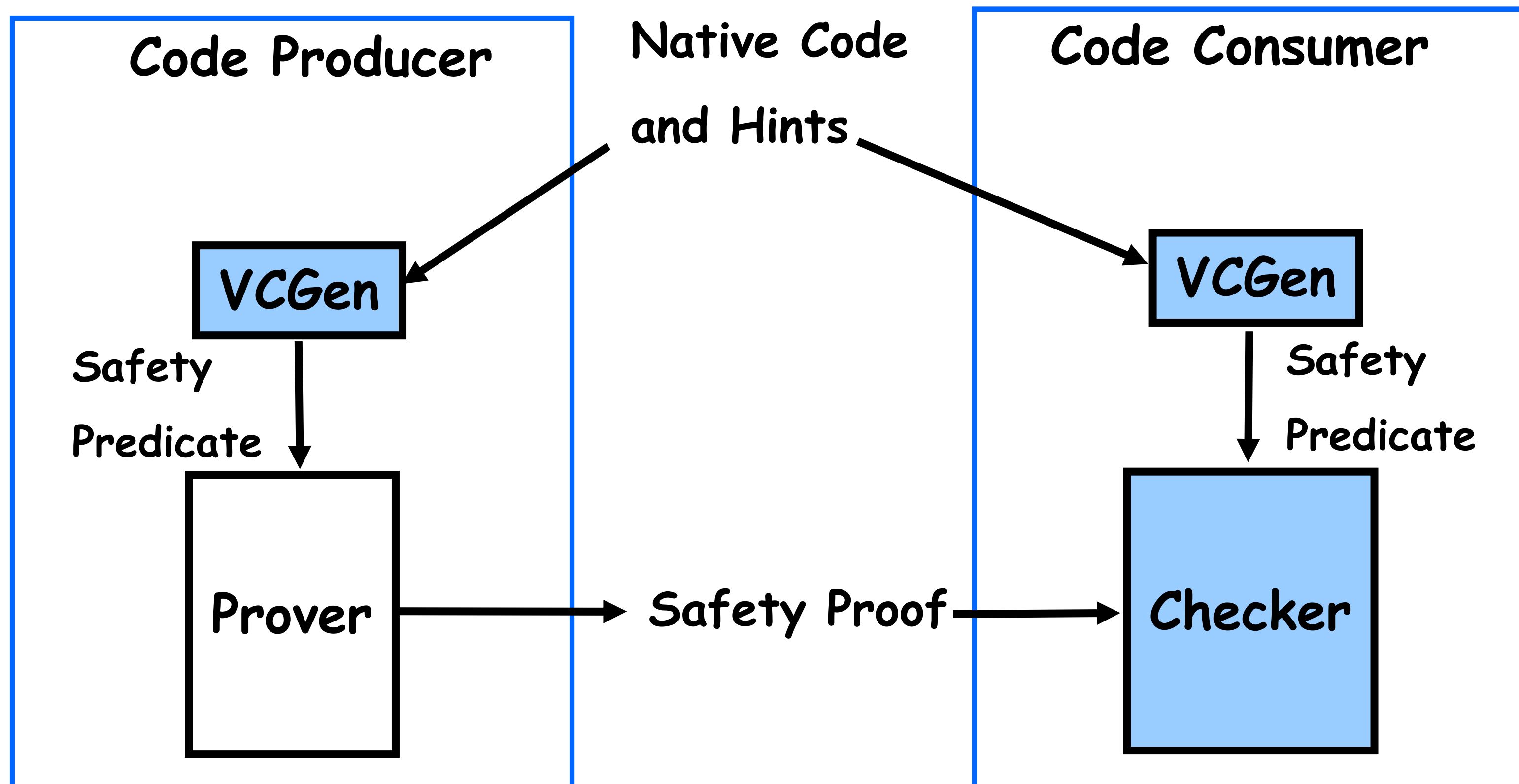
Verification Condition

- Let *Inv* be the set of line numbers containing **INV** machine instructions. Also, $0 \in Inv$.
- Inv_0 is the precondition.
- Inv_i denotes the formula at line *i*.
- *SP* is the function computing the safety predicate (verification condition) from the code.

$$SP(\Pi, Inv, post) = \bigwedge_{i \in Inv} Inv_i \rightarrow VC_{i+1}$$

Another Look at the Role of VCGen

- The VCGen provides an algorithmic way to compute the safety predicate from the native machine code instructions.
- It insures that the proof that is checked really is a proof about the code that is executed.



Correctness of VCGen Approach

- If the verification condition (safety theorem) generated from the program by VCGen is provable, then in every step of the abstract machine, a load will always be from a readable address, and a store will always be to a writable address.
- Necula & Lee [1996]

VCGen Applied to Example Program

0: $r_0 :_m intlist \wedge r_3 = 0$

⋮

2: INV $r_0 :_m intlist \wedge r_1 :_m int \wedge r_3 = 0$

⋮

9: INV $r_1 :_m int$

$$(Inv_0 \rightarrow VC_1) \wedge (Inv_2 \rightarrow VC_3) \wedge (Inv_9 \rightarrow VC_{10})$$

Computing the Last Three VCs

$VC_i := VC_{i+1}[(r_s+c)/r_d]$ *if* Π_i *is* **ADDC** $\mathbf{r_d{:}{=}r_s{+}c}$

$\quad post$ *if* Π_i *is* **RET**

$\quad p$ *if* Π_i *is* **INV** ***p***

```
    INV  r1 :m int
L2  ADDC r0 := r1 + 0   %put total in r0
    RET
```

VC_{11} is $\top$

VC_{10} is $VC_{11}[(r_1+0)/r_0] \equiv \top$

VC_9 is $r_1 :_m int$

$$(Inv_9 \to VC_{10}) \equiv (r_1 :_m int \to \top)$$

Computing the Next VC

$VC_i := (r_{s1} = r_{s2} \rightarrow VC_{i+c-1}) \wedge (\neg(r_{s1} = r_{s2}) \rightarrow VC_{i+1})$

if Π_i *is* **BEQ** $(r_{s1}=r_{s2})$ **c**

```
    INV  r0 :m intlist ∧ r1 :m int ∧ r3 = 0
L1  LD r5 := m(r0 + 0)  %r5 gets list tag
                 :
    BEQ (r3 = r3) L1    %jump back
```

VC_9 is $r_1 :_m int$

VC_8 is $(r_3 = r_3 \rightarrow VC_2) \wedge (\neg(r_3 = r_3) \rightarrow VC_9) \equiv$

$(r_3 = r_3 \rightarrow (r_0 :_m intlist \wedge r_1 :_m int \wedge r_3 = 0)) \wedge$

$(\neg(r_3 = r_3) \rightarrow (r_1 :_m int))$

Computing the Next VC

$VC_i := VC_{i+1}[(r_{s1}+r_{s2})/r_d]$ if Π_i is $\mathtt{ADD}\ \mathtt{r_d{:=}r_{s1}{+}r_{s2}}$

```
ADD r1 := r1 + r2    %add next int to total
```

VC_8 is $(r_3 = r_3 \rightarrow (r_0 :_m intlist \wedge r_1 :_m int \wedge r_3 = 0)) \wedge$

$(\neg(r_3 = r_3) \rightarrow (r_1 :_m int))$

VC_7 is $VC_8[(r_1+r_2)/r_1] \equiv$

$(r_3 = r_3 \rightarrow (r_0 :_m intlist \wedge ((r_1+r_2) :_m int) \wedge r_3 = 0)) \wedge$

$(\neg(r_3 = r_3) \rightarrow ((r_1+r_2) :_m int))$

Conjunct 2 of the Verification Condition

- Exercise: Compute $(Inv_2 \rightarrow VC_3)$
- Solution:

$$
\begin{aligned}
&(r_0 :_m intlist \wedge r_1 :_m int \wedge \ r_3 = 0) \rightarrow \\
&\quad ((m(r_0+0) = r_3 \rightarrow r_1 :_m int) \wedge \\
&\quad\; (\neg(m(r_0+0) = r_3) \rightarrow \\
&\qquad ((r_3=r_3 \rightarrow (m(r_0+2) :_m intlist \wedge (r_1 + m(r_0+1)):_m int \wedge \\
&\qquad\qquad\qquad r_3 = 0)) \wedge \\
&\qquad\; (\neg\, (r_3=r_3) \rightarrow (r_1 + m(r_0+1)):_m int) \wedge \\
&\qquad\; readable(r_0+2) \wedge \\
&\qquad\; readable(r_0+1))) \wedge \\
&\quad\; readable(r_0+0))
\end{aligned}
$$

Conjunct 1 of the Verification Condition

- Exercise: Compute $(Inv_0 \rightarrow VC_1)$
- Solution:

$$(r_0 :_m intlist \wedge r_3 = 0) \rightarrow$$
$$(r_0 :_m intlist \wedge ((r_3 + r_3) :_m int) \wedge r_3 = 0)$$

Outline: Checker

- The safety policy (inference rules) implemented by the checker includes:
  1. First-order propositional logic with natural numbers
  2. Typing rules
  3. Safety rules
  4. Interface Rules
- The Trusted Code Base (TCB) includes:
  - All these rules
  - The implementation of the checker which checks proofs built from these rules

Safety Policy: (1) Basic Logic

- Propositional logic with arithmetic (integers)
- Propositional Rules:

$$\frac{A \quad B}{A \wedge B}\wedge i \qquad \frac{A \wedge B}{A}\wedge e1 \qquad \frac{A \wedge B}{B}\wedge e2 \qquad \frac{\boxed{\begin{array}{c}A\\ \vdots \\ \bot\end{array}}}{\neg A}\neg i \qquad \frac{A \quad \neg A}{\bot}\neg e$$

$$\frac{A}{A \vee B}\vee i1 \qquad \frac{B}{A \vee B}\vee i2 \qquad \frac{A \vee B \quad \boxed{\begin{array}{c}A\\ \vdots \\ C\end{array}} \quad \boxed{\begin{array}{c}B\\ \vdots \\ C\end{array}}}{C}\vee e \qquad \frac{\bot}{A}\bot e$$

$$\frac{\boxed{\begin{array}{c}A\\ \vdots \\ B\end{array}}}{A \rightarrow B}\rightarrow i \qquad \frac{A \quad A \rightarrow B}{B}\rightarrow e \qquad \frac{\neg\neg A}{A}\neg\neg e$$

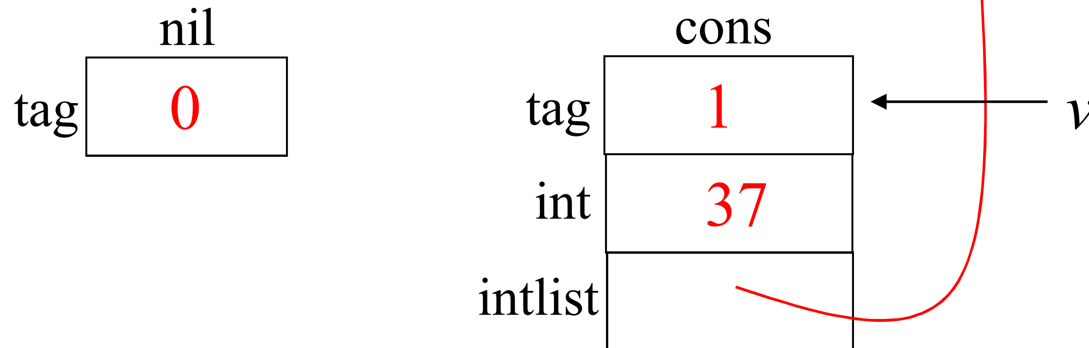
Safety Policy (2): Typing Rules

- Integers:

$$0 :_m int \qquad \frac{x :_m int}{x+1 :_m int} \qquad \frac{x :_m int \qquad y :_m int}{x+y :_m int}$$

Safety Policy (2): Typing Rules

- Integer lists:



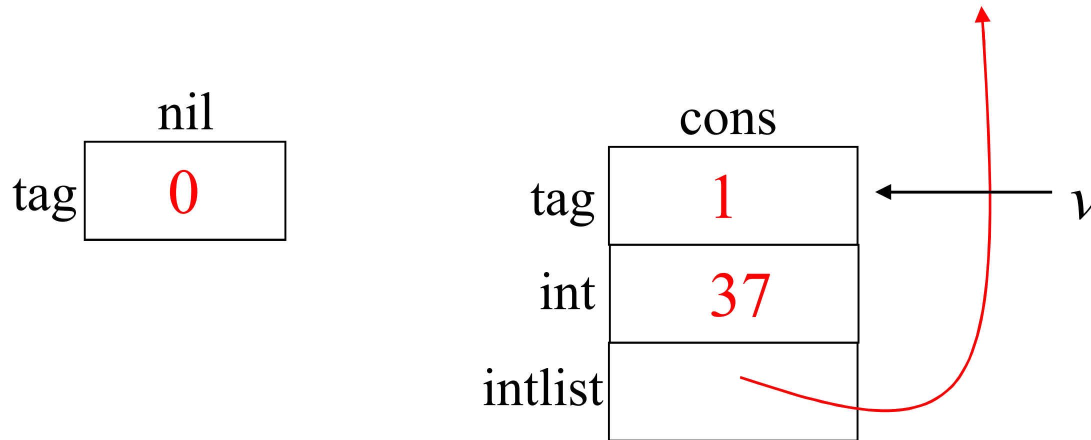
$$\frac{m(v) = 0}{v :_m intlist} \qquad \frac{m(v)=1 \quad m(v+1) :_m int \quad m(v+2) :_m intlist}{v :_m intlist}$$

$$\frac{v :_m intlist}{m(v) = 0 \vee m(v) = 1}$$

$$\frac{v :_m intlist \qquad m(v)=1}{m(v+1) :_m int} \qquad \frac{v :_m intlist \qquad m(v)=1}{m(v+2) :_m intlist}$$

Safety Policy (3)

- Safety rules, *e.g.*,



$$\frac{v :_m intlist}{readable(v)}$$

$$\frac{v :_m intlist \qquad m(v)=1}{readable(v+1)} \qquad \frac{v :_m intlist \qquad m(v)=1}{readable(v+2)}$$

Safety Policy (4)

- Interface rules: describe the calling conventions between the code consumer and the foreign code

Outline: Prover

- Safety predicates have a regular form (a small subset of formulas). This class of formulas is easy to automate.
- We don’t discuss the automated prover here.
- Instead, we show part of a proof of safety (for our running example) that is generated by such a prover.

Proof of Safety

- Part 1: Proof of $(Inv_0 \rightarrow VC_1)$

1. $(r_0 :_m intlist \wedge\ r_3 = 0)$	assump
2. $r_0 :_m intlist$	$\wedge$ *e1* 1
3. $r_3 = 0$	$\wedge$ *e2* 1
4. $0 :_m int$	typing axiom
5. $r_3 :_m int$	arith 3,4
6. $(r_3 + r_3) :_m int$	type rule 5,5
7. $(r_0 :_m intlist \wedge ((r_3 + r_3) :_m int) \wedge\ r_3 = 0)$	$\wedge$ *i* 2,6,3

8. $(r_0 :_m intlist \wedge\ r_3 = 0) \rightarrow (r_0 :_m intlist \wedge ((r_3 + r_3) :_m int) \wedge\ r_3 = 0)$

Outline: Executing the Program

- One step left: checking the precondition

Checking the Precondition

- After checking the proof, the program can be executed as many times as needed, but the precondition must be checked (proved) each time.
- For this example, there is a simple proof of:

  $r_0 :_m intlist \wedge r_3 = 0$