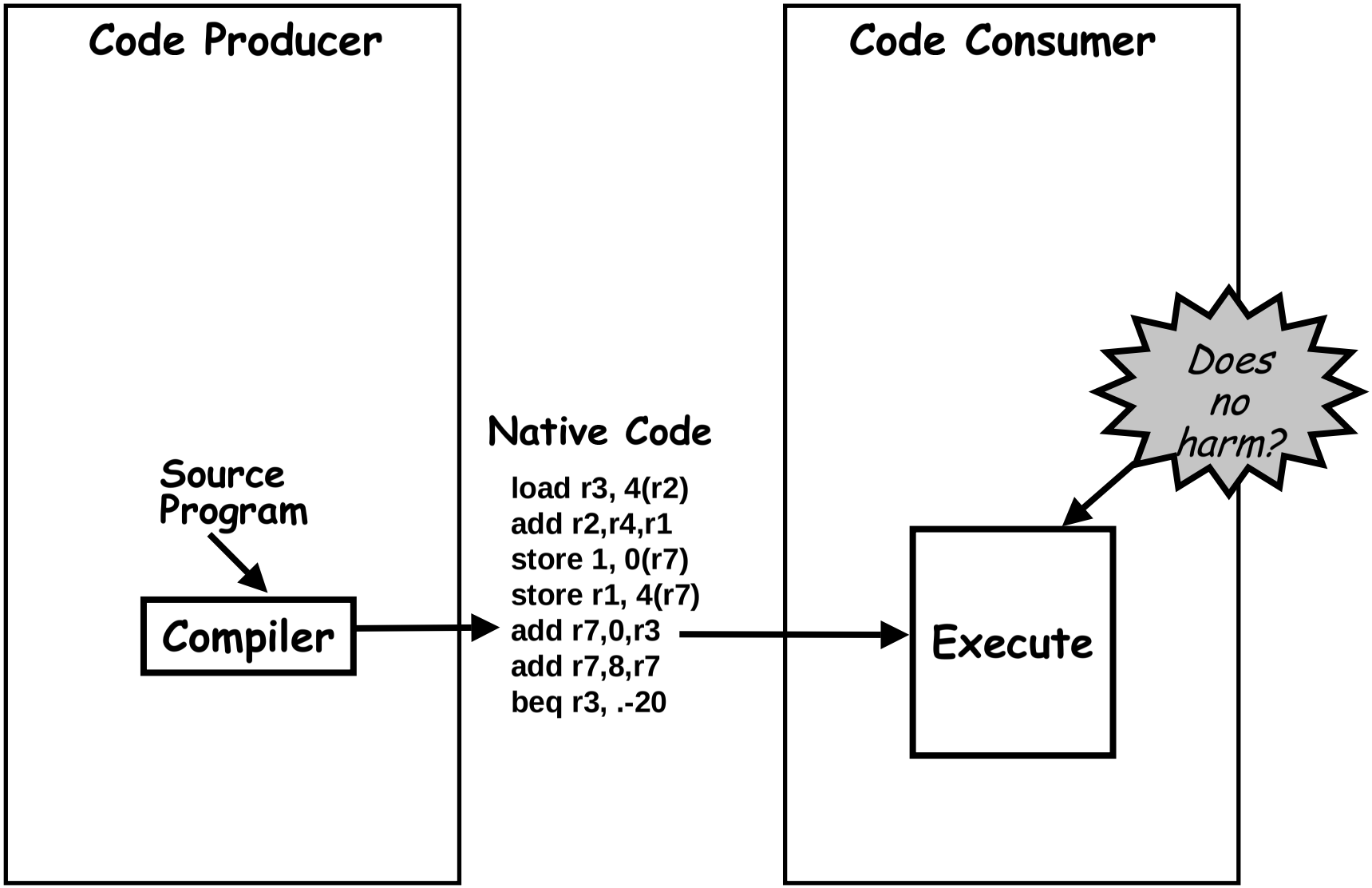
An Introduction to Proof-Carrying Code

Amy Felty
University of Ottawa
June 2002

The Problem: Code Safety



References

- Part I: Proof-Carrying Code
  - **www.cs.berkeley.edu/~necula/papers.html**
  - **www-2.cs.cmu.edu/~fox/pcc.html**
  - George Necula & Peter Lee, Proof-Carrying Code, Technical Report CMU-CS-96-165, 1996.
  - 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.
- Part II: Foundational Proof-Carrying Code
  - **www.cs.princeton.edu/sip/projects/pcc**
  - Andrew Appel & Amy Felty, A Semantic Model of Types and Machine Instructions for Proof-Carrying Code, Symposium on Principles of Programming Languages (POPL), 2000.
  - Andrew Appel, Foundational Proof-Carrying Code, Symposium on Logic in Computer Science (LICS), 2001.

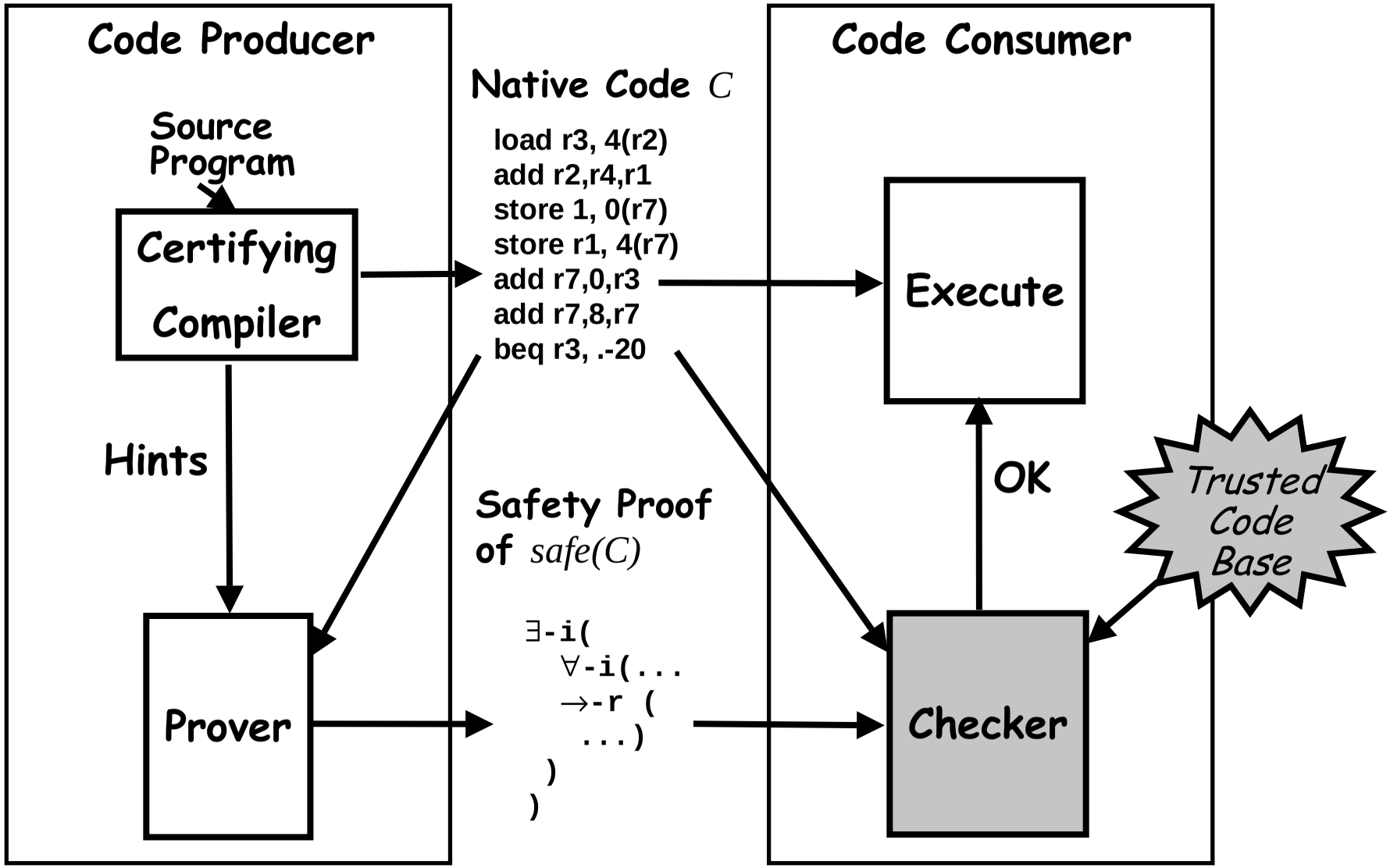
Proof-Carrying Code

Code Producer

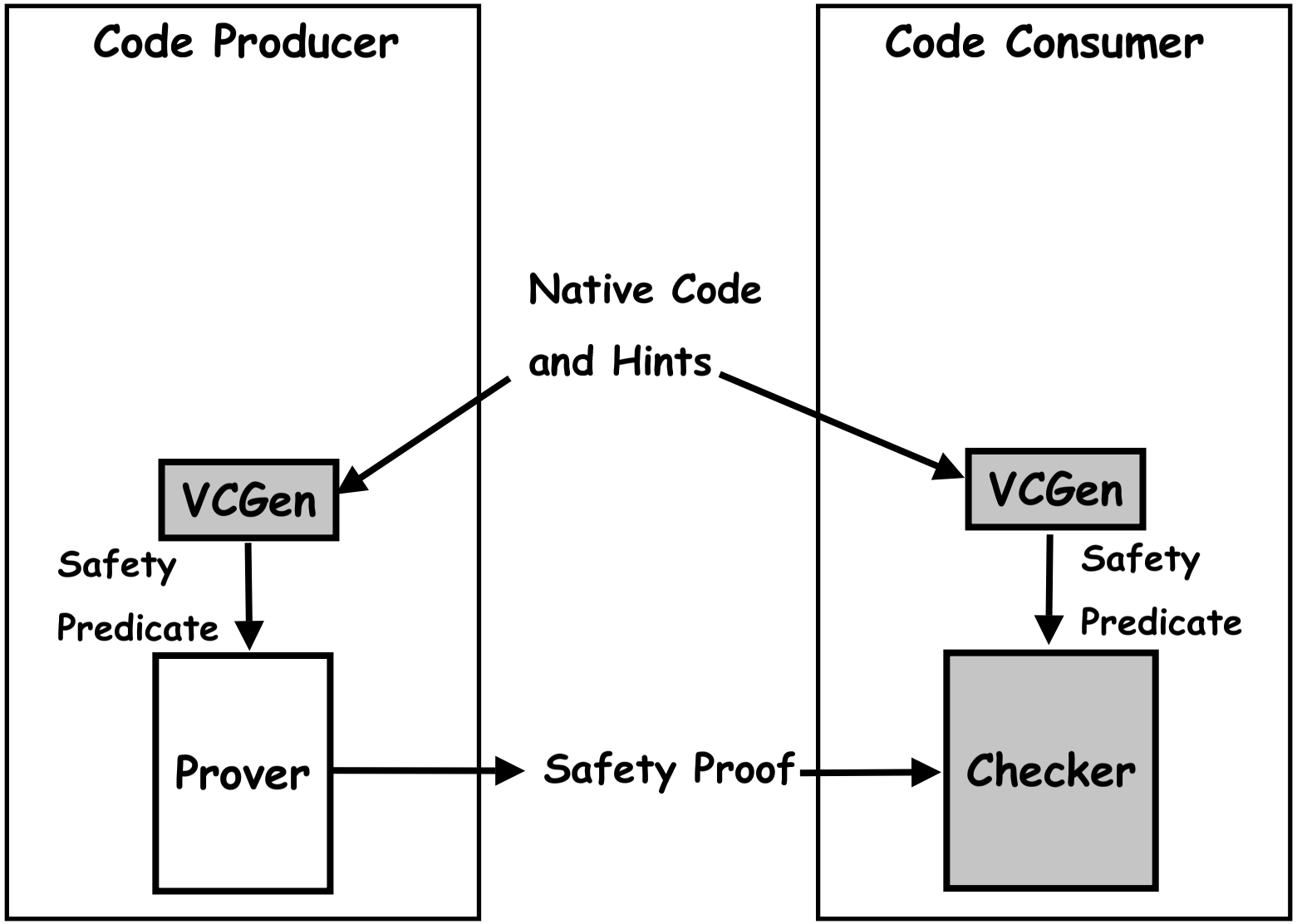
- Implements a program and compiles it to native machine code *C*.
- The verification condition *safe(C)* is sent to a prover which proves it (automatically) and outputs a proof *P*.
- The compiler also sends hints to the prover.

- 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

Part I: 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

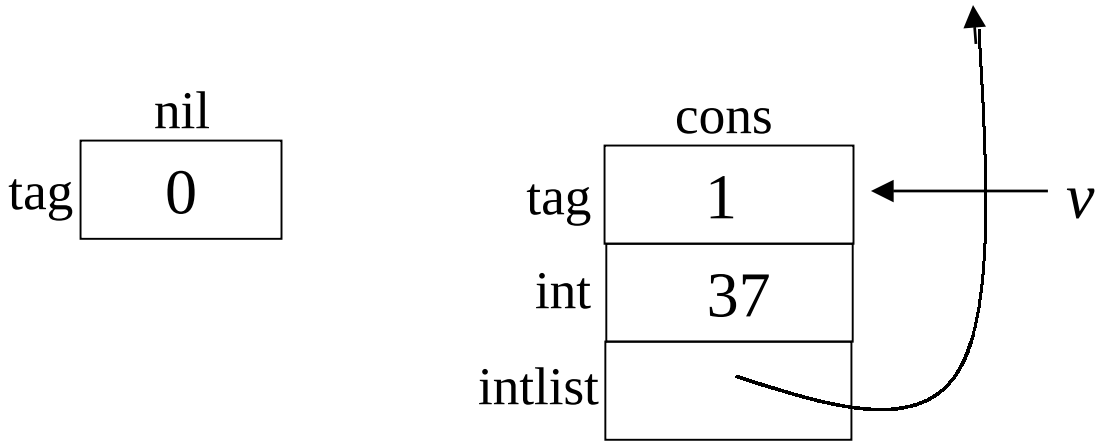
```
datatype intlist =
    nil of ()   |   cons of int * intlist;

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 (first-order logic)
- The abstract machine
  - Needed to prove soundness of PCC
- 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_{s2} + c) := r_{s1}**
- **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 ::== $x \mid n \mid e_1 + e_2 \mid r_n \mid m(e)$

Formulas of Safety Policy

- Types:

  τ ::== *int* | *intlist* | …

- Predicates:

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

- Formulas:

  P ::== $A \mid true \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \Rightarrow P_2 \mid \neg P \mid$
  $\forall xP \mid \exists xP$

The Abstract Machine: One Computation Step

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

if $(upd(r,pc, r_{pc} + 1),m)$ evaluates to (r',m')
by executing one instruction

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_{s2} + c) := r_{s1} *and writable(r_{s2} + c)*	*r*	*upd(m, r_{s2} + c, r_{s1})*
RET	*r*	*m*
INV p	*r*	*m*

The Abstract Machine (2)

(r,m) evaluates to:

Instruction	*r'*	*m'*
BEQ (r_{s1} = r_{s2}) c	$upd(r,pc, r_{pc} + c - 1)$	*m*
and $(r_{s1} = r_{s2})$		
BEQ (r_{s1} = r_{s2}) c	*r*	*m*
and $(r_{s1} \neq r_{s2})$		
BGT (r_{s1} > r_{s2}) c	$upd(r,pc, r_{pc} + c - 1)$	*m*
and $(r_{s1} > r_{s2})$		
BGT (r_{s1} > r_{s2}) c	*r*	*m*
and $(r_{s1} \leq r_{s2})$		

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
- Soundness of VCGen
- Example safety predicate generated by the algorithm

The Verification Condition

- A predicate in first-order 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

$\{[e/x]Q\}\ x:=e\ \{Q\}$ $\qquad$ *e.g.*, $\{x=0\}$ **`y:=0`** $\{x=y\}$

$$\frac{\{P \wedge B\}S_1\{Q\} \qquad \{P \wedge \neg B\}S_2\{Q\}}{\{P\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\{Q\}}$$

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

$$\frac{P \Rightarrow P' \quad \{P'\}S\{Q'\} \quad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

Hoare Logic for Machine Instructions (1)

$$\frac{Pre \Rightarrow Q_0 \quad \{Q_0\}S_1\{Q_1\} \quad \{Q_1\}S_2\{Q_2\} \ldots \{Q_{n-1}\}S_n\{Q_n\} \quad Q_n \Rightarrow Post}{\{Pre\}S_1;S_2;\ldots;S_n\{Post\}}$$

$$\frac{P \Rightarrow P' \quad \{P'\}S\{Q'\} \quad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

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

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

Hoare Logic for Machine Instructions (2)

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

$\{[upd(m,r_{s2}+c,r_{s1})/m]Q \wedge writable(r_{s2}+c)\}$ **ST m(r_{s2}+c):=r_{s1}** $\{Q\}$

$\{(r_{s1} = r_{s2} \Rightarrow Q_c) \wedge (\neg(r_{s1} = r_{s2}) \Rightarrow Q)\}$ **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.

.

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

Definition of VCG (continued)

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

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

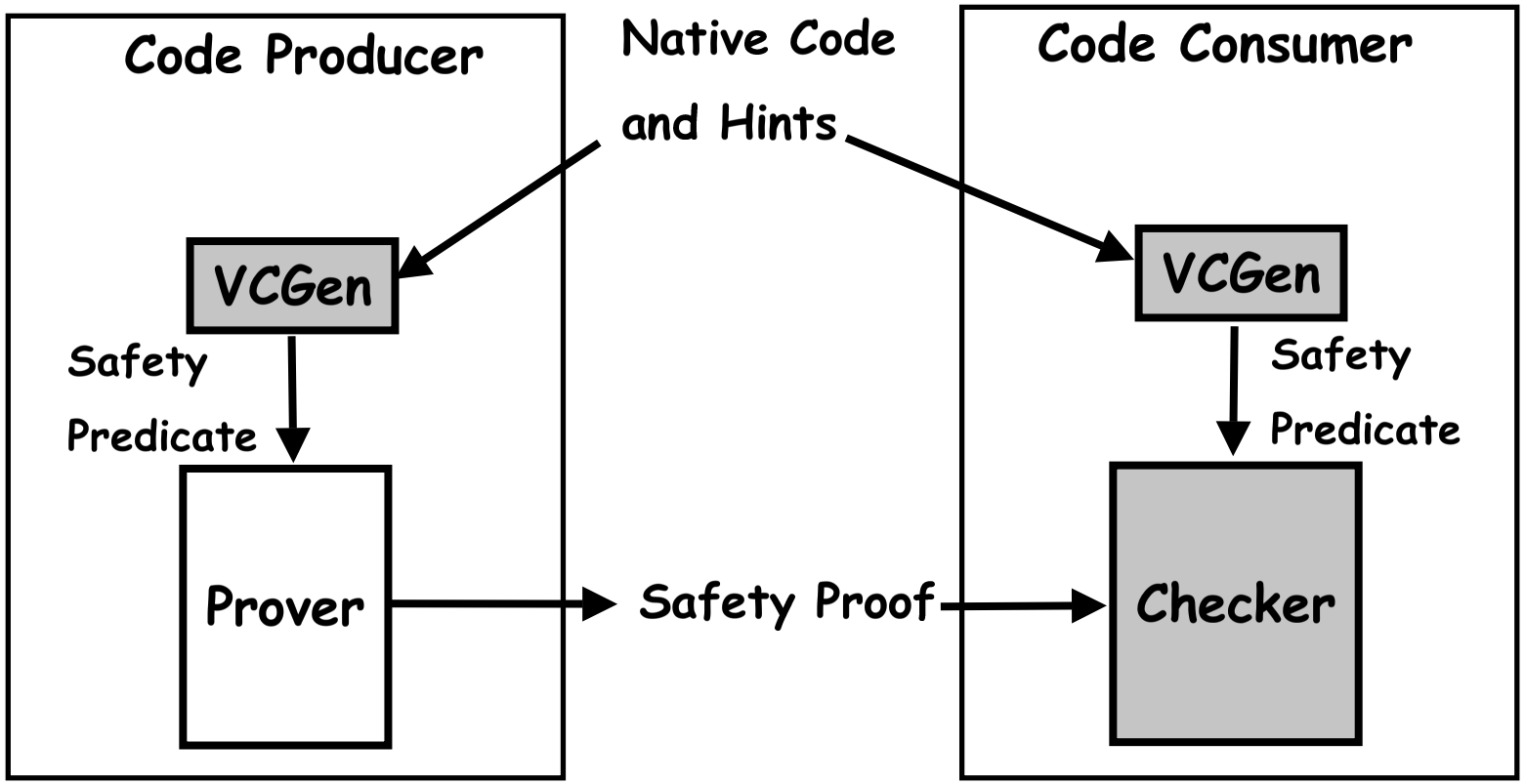
Verification Condition

- 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.

$$SP(\Pi, Inv, post) = \forall k \ \forall r_k \ \bigwedge_{i \in Inv} Inv_i \Rightarrow VC_{i+1}$$

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.



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

$\vdots$

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

$\vdots$

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 First Three VCs

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

post *if* Π_i *is* **RET**

p *if* Π_i *is* **INV *p***

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

VC_{11} is *true*

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

VC_9 is $r_1 :_m$ *int*

$$(Inv_9 \Rightarrow VC_{10}) \equiv (r_1 :_m int \Rightarrow true)$$

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

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

ADD r_1 := r_1 + r_2 %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 $[(r_1+r_2)/r_1]VC_8 \equiv$
$(r_3 = r_3 \Rightarrow (r_0 :_m intlist \wedge ((r_1+r_2) :_m int) \wedge r_3 = 0)) \wedge$
$(\neg(r_3 = r_3) \Rightarrow ((r_1+r_2) :_m int))$

Conjunct 2 of the Verification Condition

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

$$(r_0 :_m intlist \wedge r_1 :_m int \wedge\ r_3 = 0) \Rightarrow$$
$$((m(r_0+0) = r_3 \Rightarrow r_1 :_m int) \wedge$$
$$(\neg(m(r_0+0) = r_3) \Rightarrow$$
$$((r_{3=}r_3 \Rightarrow (r_0 :_m intlist \wedge (r_1 +(r_0+1)):_m int \wedge\ r_3 = 0)) \wedge$$
$$(\neg\ (r_{3=}r_3) \Rightarrow (r_1 +(r_0+1)):_m int) \wedge$$
$$readable(m(r_0)+2) \wedge$$
$$readable(m(r_0)+1)))$$
$$readable(m(r_0)+0))$$

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 predicate logic with natural numbers and induction
  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

- First-order predicate logic with natural numbers and induction
- Propositional Rules:

$$\frac{A \quad B}{A \wedge B}\,\wedge\mathrm{I} \qquad \frac{A \wedge B}{A}\,\wedge\mathrm{E1} \qquad \frac{A \wedge B}{B}\,\wedge\mathrm{E2} \qquad \frac{\begin{array}{c}(A)\\ \bot\end{array}}{\neg A}\,\neg\mathrm{I} \qquad \frac{A \quad \neg A}{\bot}\,\neg\mathrm{E}$$

$$\frac{A}{A \vee B}\,\vee\mathrm{I1} \qquad \frac{B}{A \vee B}\,\vee\mathrm{I2} \qquad \frac{A \vee B \quad \begin{array}{c}(A)\\ C\end{array} \quad \begin{array}{c}(B)\\ C\end{array}}{C}\,\vee E \qquad \frac{\bot}{A}\,\bot E$$

$$\frac{\begin{array}{c}(A)\\ B\end{array}}{A \Rightarrow B}\,\Rightarrow\mathrm{I} \qquad \frac{A \quad A \Rightarrow B}{B}\,\Rightarrow\mathrm{E} \qquad \frac{\begin{array}{c}(\neg A)\\ \bot\end{array}}{A}$$

Safety Policy: (1) Basic Logic (continued)

- Quantifiers, equality, induction, arithmetic

$$\frac{[y/x]A}{\forall xA}\ \forall\mathrm{I} \qquad \frac{\forall xA}{[t/x]A}\ \forall\mathrm{E} \qquad \frac{[t/x]A}{\exists xA}\ \exists\mathrm{I} \qquad \frac{\exists xA \qquad \begin{matrix}([y/x]A)\\ C\end{matrix}}{C}\ \exists E$$

$$\frac{t_1=t_2 \qquad [t_1/x]A}{[t_2/x]A}$$

$$\frac{[0/x]A \qquad [n/x]A \Rightarrow [(n+1)/x]A}{\forall xA}$$

$$(x+y)+z=x+(y+z) \qquad x+y=y+x \qquad x+0=x \qquad \neg(0=x+1)$$

…

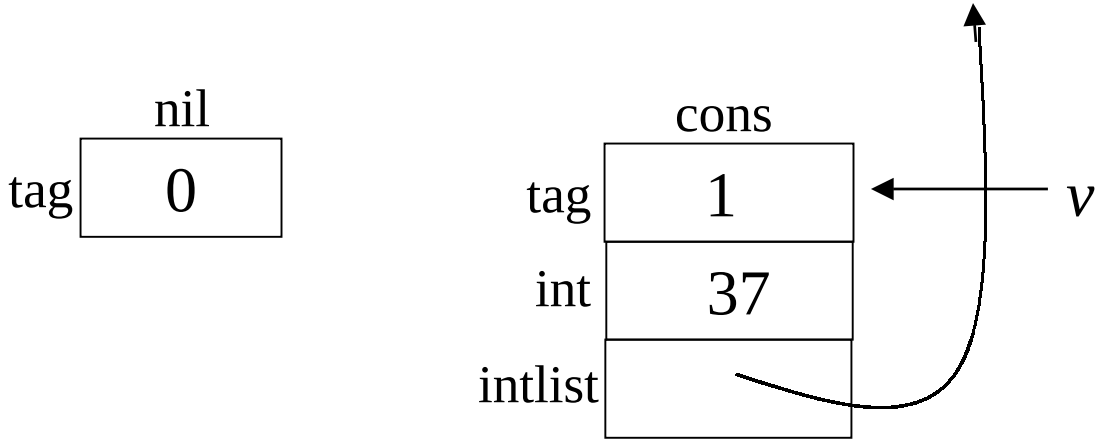
Safety Policy (2): Typing Rules

- 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:

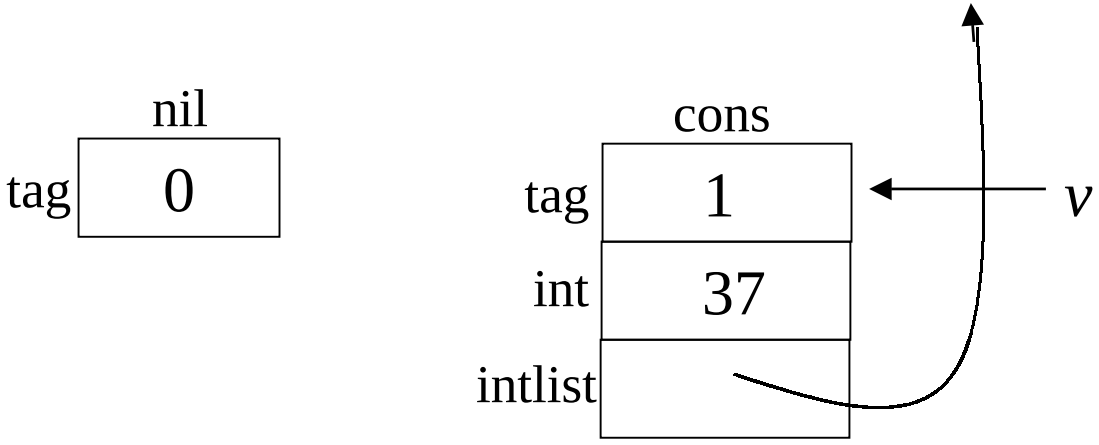


$$\frac{v :_m \mathit{intlist}}{m(v) = 0 \vee m(v) = 1}$$

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

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 first-order 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 $(Inv_0 \Rightarrow VC_1)$

$$\dfrac{\dfrac{r_0 :_m intlist \wedge\ r_3 = 0}{r_3 = 0} \qquad \dfrac{}{0 :_m int}}{r_3 :_m int}$$

$$\dfrac{\dfrac{\dfrac{r_0 :_m intlist \wedge\ r_3 = 0}{r_0 :_m intlist} \qquad \dfrac{r_3 :_m int \qquad r_3 :_m int}{(r_3 + r_3) :_m int} \qquad \dfrac{r_0 :_m intlist \wedge\ r_3 = 0}{r_3 = 0}}{r_0 :_m intlist \wedge ((r_3 + r_3) :_m int) \wedge\ r_3 = 0}}{(r_0 :_m intlist \wedge\ r_3 = 0) \Rightarrow (r_0 :_m intlist \wedge ((r_3 + r_3) :_m int) \wedge\ r_3 = 0)}$$

Outline: Executing the Program

- One step left: checking the precondition

Checking the Precondition

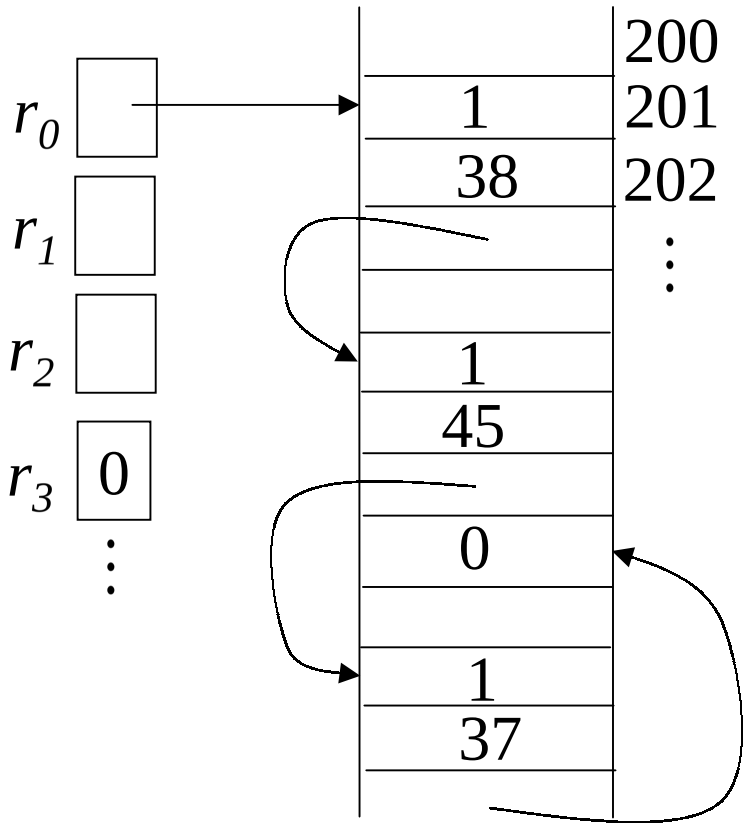
- After checking the proof, the program can be executed as many times as needed, but the precondition must be checked each time.

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

1. $\forall v(v :_m intlist \Rightarrow readable(v))$ precondition satisfied

2. $\forall v(v>200 \Rightarrow readable(v))$ precondition satisfied

3. $\forall v(v>210 \Rightarrow readable(v))$ precondition not satisfied



A Prototype Implementation

- Implemented in the Twelf system, which implements the Logical Framework.
- The Twelf files encode the safety policy and implement the checker for it.