Foundational Proof-Carrying Code

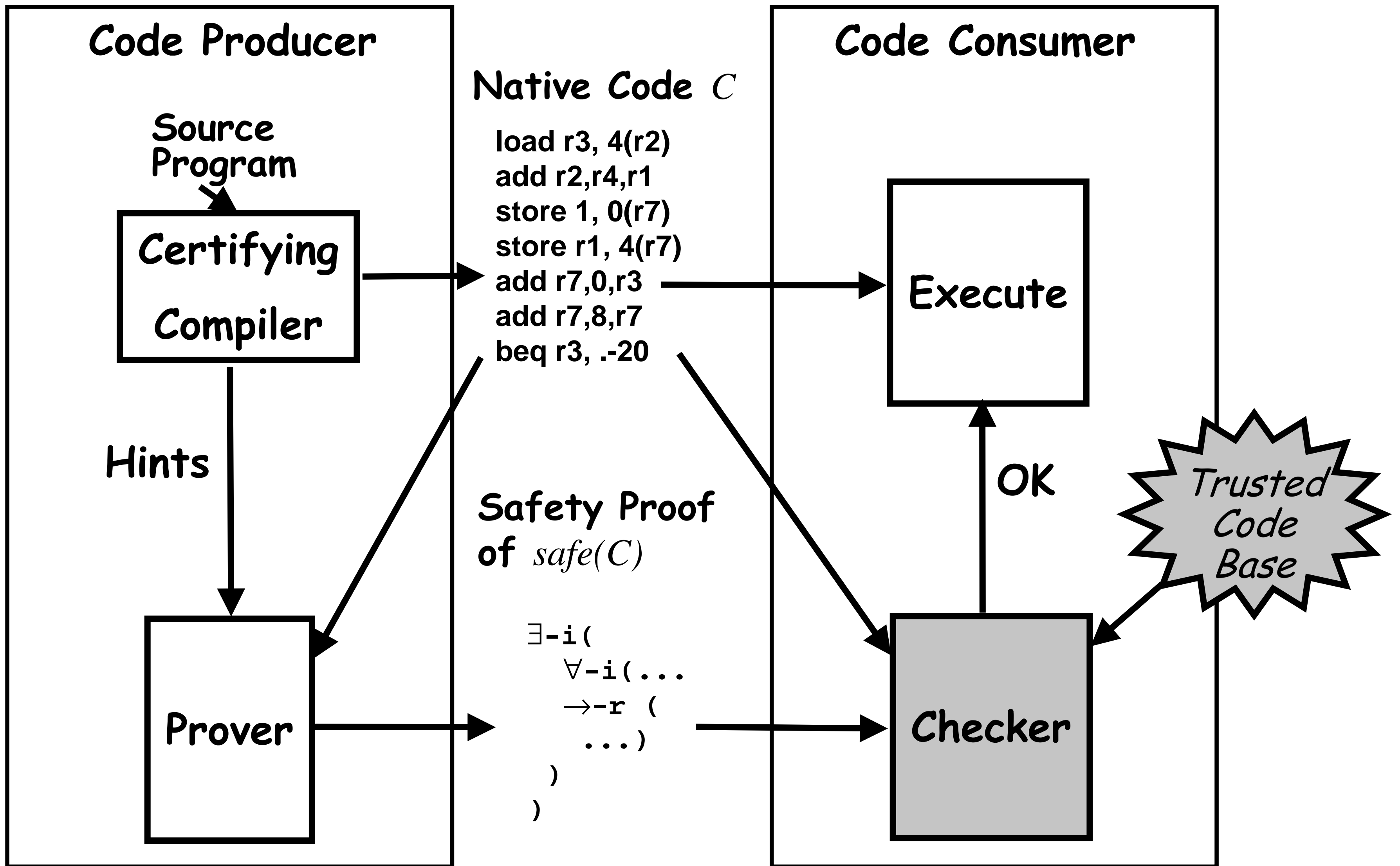
Amy Felty
University of Ottawa

June 2002

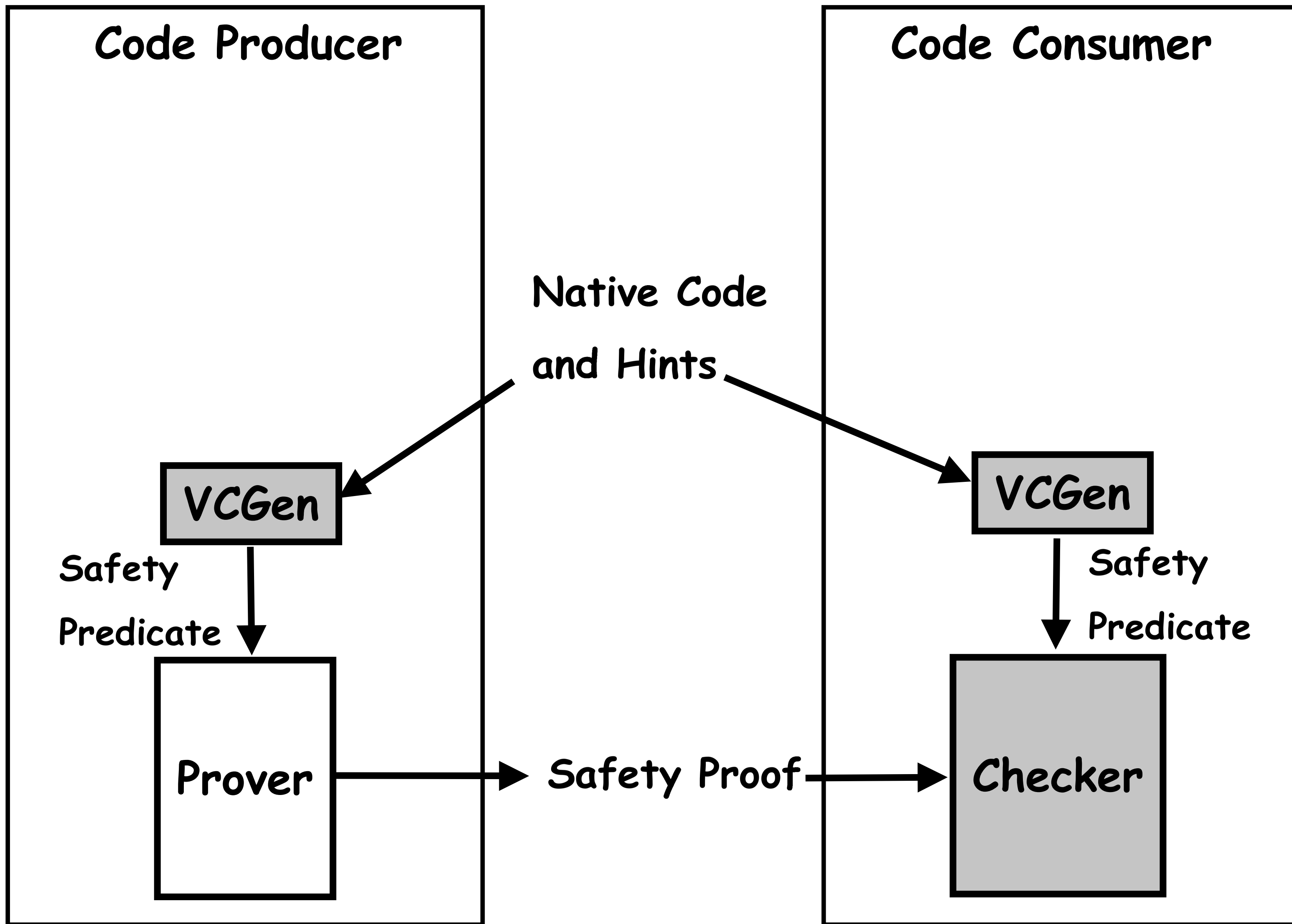
References

- 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



A Closer Look at the Prover and Checker



Safety Policy Summary

- First-order predicate logic with natural numbers and induction,
  e.g., $\dfrac{[y/x]A}{\forall xA}\,\forall I \quad \dfrac{\forall xA}{[t/x]A}\,\forall E \quad \dfrac{A \quad A \Rightarrow B}{B}\,\Rightarrow E \quad \dfrac{\begin{matrix}(A)\\ B\end{matrix}}{A \Rightarrow B}\,\Rightarrow I$
- Safety policy rules, *e.g.*,

  $\forall v(v>200 \Rightarrow readable(v))$

- Typing rules, *e.g.*,

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

- Interface rules
- The VCG expressing the semantics of machine instructions,
  e.g., $\{[m(r_s+c)/r_d]Q \wedge readable(r_s+c)\}$ **LD** $\mathbf{r_d := m(r_s+c)}$ $\{Q\}$

Foundational Proof-Carrying Code

- New approach:
  - Prove typing rules from first principles.
    - Avoids commitment to a particular type system.
    - Removes rules from safety policy.
  - Specify machine semantics directly.
    - Avoids using a Verification Condition Generator.
- Advantages:
  - **Increased Security:** The Trusted Code Base (TCB), i.e., the proof checker, is smaller.
  - **Increased Flexibility:** Allows programs compiled from different source languages to be sent to the same code consumer.

Safety Policy, New Version

- A higher-order logic (simple theory of types [Church, JSL'40]) with integers and natural number induction
- Safety rules: remove those about types, others unchanged
- Typing rules removed
- Interface rules
- Replace VCGen with direct encoding of machine semantics

Part II: Outline

- New basic rules
- Removing typing rules (and safety rules about types)
  - Encoding Types
  - Handling allocation
- Encoding Machine Instructions

Higher-Order Logic

Higher-order logic with natural numbers and induction

Types and terms of the simply typed λcalculus

- Types:
  - Formulas have type *o*.
  - Register numbers, register contents, addresses and memory contents all have type *num*.
  - Also all other types $\tau_1 \rightarrow \tau_2$ such that τ_1 and τ_2 are types.
- Terms:
  - Variables: x
  - Application: $(t_1\ t_2)$
  - Abstraction: $\lambda x{:}\tau.t$

Functions and Predicates

- Functions:
  - $+ : num \rightarrow num \rightarrow num$

    $\vdots$

- Predicates:
  - $=_\tau : \tau \rightarrow \tau \rightarrow o$ for every type τ
  - $< : num \rightarrow num \rightarrow o$ for every type τ

    $\vdots$

- Note: predicates always have types of the form $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow o$

Formulas of Higher-Order Logic

- Implication:
  - $\Rightarrow : o \rightarrow \mathrm{o} \rightarrow \mathrm{o}$
  - Written as infix as usual: $A \Rightarrow B$
- Universal quantification:
  - $\forall_\tau : (\tau \rightarrow o) \rightarrow \mathrm{o}$ for every type τ
  - $(\forall_\tau\, \lambda x{:}\tau.A)$ abbreviated as $\forall x : \tau.A$ or $\forall x.A$
  - Note: quantification can be over objects of any type, including function types and predicates.

Inference Rules (1)

- Basic rules

$$\frac{\begin{matrix}(A)\\ B\end{matrix}}{A \Rightarrow B} \Rightarrow I \qquad \frac{A \quad A \Rightarrow B}{B} \Rightarrow E$$

$$\frac{\begin{matrix}(y{:}\tau)\\ [y/x]A\end{matrix}}{\forall_\tau x.A} \forall I \qquad \frac{\forall_\tau xA \qquad t{:}\tau}{[t/x]A} \forall E$$

$$(\lambda x{:}\tau.t_1)t_2 =_\tau [t_1/x]t_2 \qquad \beta$$

Inference Rules (2)

- Natural numbers and induction as before

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

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

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

$$\ldots$$

Definitions for Other Connectives

- We use the following definitions for the other connectives.
- The logic in our safety policy is smaller, but it now allows definitions.

- $\wedge \equiv \lambda A{:}o.\ \lambda B{:}o.\ \forall_o\, C.(A \Rightarrow B \Rightarrow C) \Rightarrow C$
- $\vee \equiv \lambda A{:}o.\ \lambda B{:}o.\ \forall_o\, C.(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow C$
- $\bot \equiv \forall_o\, A.A$
- $\neg \equiv \lambda A{:}o..(A \Rightarrow \bot)$
- $\exists \equiv \lambda F{:}\tau \rightarrow \mathrm{o}.\ \forall_o\, B.(\forall_\tau\, x.((F\ x) \Rightarrow\ B) \Rightarrow B$

A Lemma

- Lemma

$$\frac{A \quad B}{A \wedge B}\wedge I$$

- Proof

$$\cfrac{\cfrac{\cfrac{B \qquad \cfrac{A \qquad A \Rightarrow B \Rightarrow C}{B \Rightarrow C}}{C}}{(A \Rightarrow B \Rightarrow C) \Rightarrow C}}{\cfrac{\forall_o C.(A \Rightarrow B \Rightarrow C) \Rightarrow C}{A \wedge B}}$$

Lemmas in Proofs

- Such definitions and lemmas become part of every proof that uses these connectives.
- Proofs become bigger, but a large part of each proof of safety is now a fixed set of lemmas which can be checked once and for all.

Other Inference Rules

- Derived Rules (Lemmas)

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

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

- Primitive Rule (for classical logic)

$$\frac{\begin{array}{c}(\neg A)\\ \bot\end{array}}{A}$$

Typing Rules

- Built-in rules
  - Lock code producer into a predetermined programming language.
  - Also force a predetermined field-layout (predetermined compiler).
- Foundational approach
  - Each type is a defined predicate.
  - Each typing rule is a lemma proved from the definitions.
  - Proofs of lemmas can be incorporated into proof sent by code producer.
  - Not relying on metatheorems, *e.g.*, soundness of typing rules.

Functions in Higher-Order Logic for PCC (1)

Register bank:

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

- $r : num \rightarrow num$
- $upd : (num \rightarrow num) \rightarrow num \rightarrow num \rightarrow (num \rightarrow num) \rightarrow o$

- Memory:

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

- $m : num \rightarrow num$

Functions in Higher-Order Logic for PCC (2)

- Expressions:

$$e ::== \; x \mid n \,/\, e_1 + e_2 \,/\, r_n \,/\, m(e)$$

  - $e : num$
  - $+ : num \rightarrow num \rightarrow num$
  - $r : num \rightarrow num$
  - $m : num \rightarrow num$

Predicates in Higher-Order Logic

- Predicates:

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

  - $readable : num \rightarrow o$
  - $writable: num \rightarrow o$
  - Instead of a the typing judgment $(e :_m \tau)$ as a predicate of 3 arguments (e, m, and τ), types at the object level (of the programming language) are now predicates of 2 arguments.

    $\tau : (num \rightarrow num) \rightarrow num \rightarrow o$

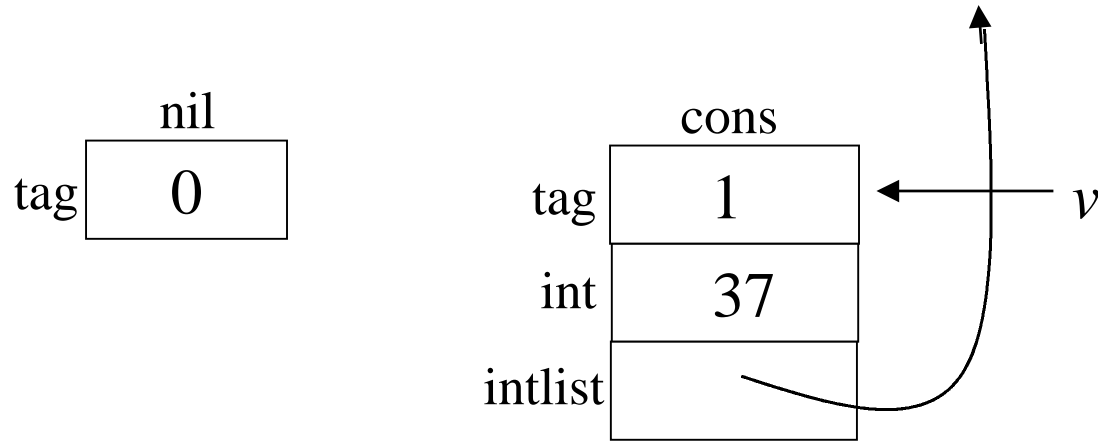
  We now write $(\tau\, m\, e)$ instead of $(e :_m \tau)$.

  $int : (num \rightarrow num) \rightarrow num \rightarrow o$

  $intlist : (num \rightarrow num) \rightarrow num \rightarrow o$

Integer Lists Revisited (1)

- Integer lists:

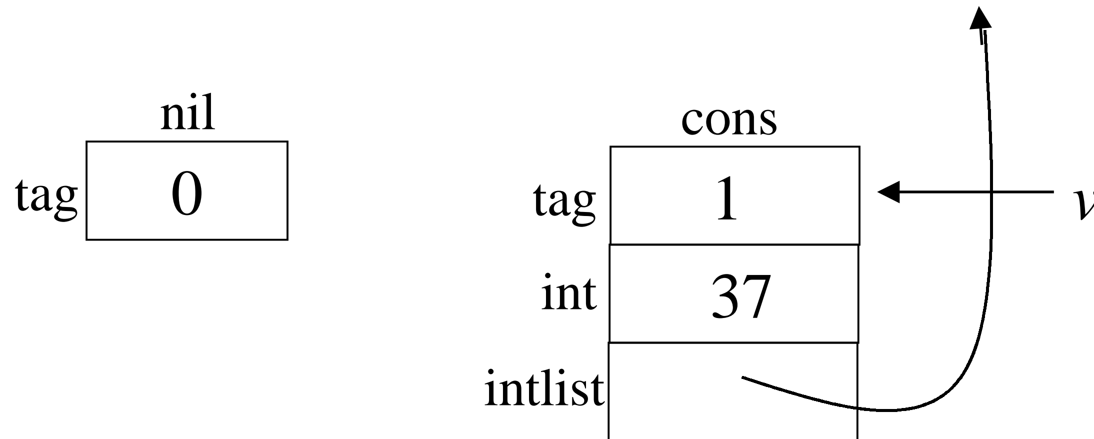


$$\frac{v :_m intlist}{m(v) = 0 \vee m(v) = 1}$$

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

Integer Lists Revisited (2)

- Safety rules, *e.g.*,



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

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

Types as Definitions

- Integer lists defined as:

$$\begin{aligned} intlist \equiv \lambda m\ \lambda v.\ ((&readable\ v) \wedge \\ &[(m\ v) = 0 \vee \\ &\quad ((m\ v) = 1 \wedge (readable\ (v+1)) \wedge (int\ m\ (v+1)) \wedge \\ &\quad\ (readable\ (v+2)) \wedge (intlist\ m\ (v+2))] \end{aligned}$$

- Six rules for accessing integer lists now derivable as lemmas?

e.g., $\dfrac{v :_m intlist}{readable(v)} \qquad \dfrac{v :_m intlist \qquad m(v)=1}{m(v+2) :_m intlist}$

- Two problems: (1) recursive definitions, and (2) allocation.

Recursive Datatypes Details

- Let $tp \equiv (num \rightarrow num) \rightarrow num \rightarrow o$
- Subtypes and the *rec* operator.

$$subtype : tp \rightarrow tp \rightarrow o$$
$$rec : (tp \rightarrow tp) \rightarrow (num \rightarrow num) \rightarrow num \rightarrow o$$

$$subtype \equiv \lambda \tau_1, \tau_2.\ (\forall m,v.\ ((\tau_1\ m\ v) \Rightarrow (\tau_2\ m\ v)))$$
$$rec \equiv \lambda f,m,v.(\forall \tau.\ ((subtype\ (f\ \tau)\ \tau) \Rightarrow (\tau\ m\ v)))$$

- The recursive types are all types *(rec f)* for which the least fixed point of the argument function *f* is *(rec f)* (the **fold/unfold** property).

$$(rec\ f\ m\ v) \Leftrightarrow (f\ (rec\ f)\ m\ v)$$

Another Lemma

- This property holds of all functions f that are monotone:

$$monotone : (tp \rightarrow tp) \rightarrow o$$

$$monotone \equiv$$

$$\lambda f.\ (\forall \tau_1, \tau_2.\ ((subtype\ \tau_1\ \tau_2) \Rightarrow (subtype\ (f\ \tau_1)\ (f\ \tau_2))))$$

- We prove:

$$\forall f{:}(tp \rightarrow tp).\ \forall m{:}(num \rightarrow num).\ \forall v{:}num$$

$$(monotone\ f) \Rightarrow ((rec\ f\ m\ v) \Leftrightarrow (f\ (rec\ f)\ m\ v))$$

Recursive Datatypes Summary

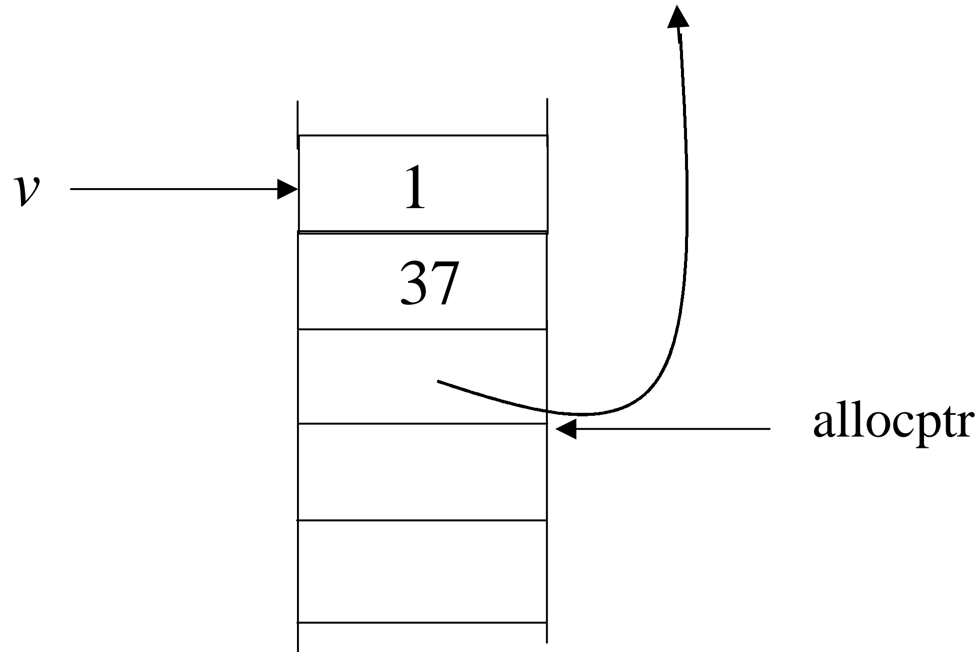
- Programming languages provide syntax for user-defined recursive datatypes.
- To reason about them, we have to get the semantics right.
- Higher-order logic provides us with the tools to express these semantics.
- In general, inductive reasoning about recursively defined objects requires monotone operators.

Recursive Integer Lists

$$\begin{aligned}
&intlist \equiv rec\ (\lambda\tau.\lambda m.\lambda v. \\
&\quad [(readable\ v) \wedge \\
&\quad ((m\ v) = 0 \vee \\
&\quad ((m\ v) = 1 \wedge \\
&\quad (readable\ (v{+}1)) \wedge (int\ m\ (m\ (v{+}1))) \wedge \\
&\quad (readable\ (v{+}2)) \wedge (\tau\ m\ (m\ (v{+}2)))))])
\end{aligned}$$

$$\begin{aligned}
&monotone\ (\lambda\tau.\lambda m.\lambda v. \\
&\quad [(readable\ v) \wedge \\
&\quad ((m\ v) = 0 \vee \\
&\quad ((m\ v) = 1 \wedge \\
&\quad (readable\ (v{+}1)) \wedge (int\ m\ (m\ (v{+}1))) \wedge \\
&\quad (readable\ (v{+}2)) \wedge (\tau\ m\ (m\ (v{+}2)))))])
\end{aligned}$$

Allocation

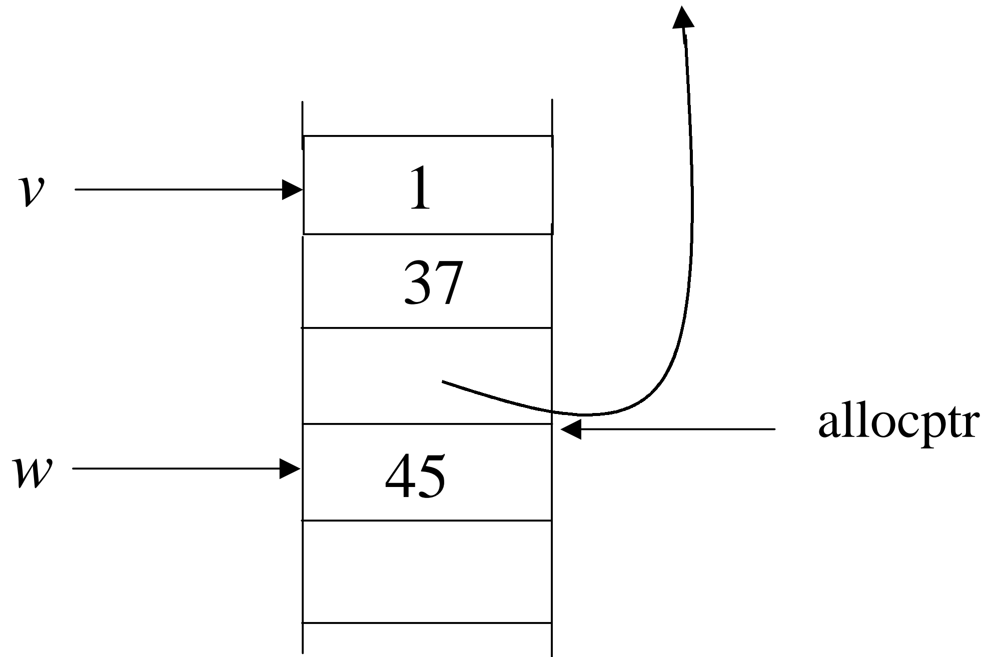


- An *allocptr* is used to keep track of the boundary between allocated (above) and unallocated (below) memory locations.
- If allocated memory at address v has fields with the right properties, then v has type *intlist*.

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

Allocation (continued)

- Allocating new data doesn't affect the types of old data.



$$\frac{v :_m \tau \qquad writable(w)}{v :_{m[w \to u]} \tau}$$

Incorporating Allocation

- The typing judgment is parameterized by an **allocation predicate**: $(\tau\ A\ m\ v)$.
- The *valid* predicate encompasses **initialization invariance** and **allocation invariance**.

$$
\begin{aligned}
& \mathit{valid} \equiv \\
& \quad \lambda\tau.\ (\forall m,v,A,w,u,m'. \\
& \qquad\qquad (((\tau\ A\ m\ v) \wedge \neg(A\ w) \wedge (\mathit{upd}\ m\ w\ u\ m')) \Rightarrow (\tau\ A\ m'\ v)) \wedge \\
& \qquad\quad \forall m,v,A,A'. \\
& \qquad\qquad (((\tau\ A\ m\ v) \wedge (\forall z.(A\ z \Rightarrow A'\ z))) \Rightarrow (\tau\ A'\ m\ v)))
\end{aligned}
$$

Integer Lists and Allocation

- A revised *intlist:*

$$
\begin{aligned}
intlist \equiv\ & rec\ (\lambda\tau.\ \lambda A.\ \lambda m.\ \lambda v. \\
& [(readable\ v) \wedge (A\ v) \wedge \\
& ((m\ v) = 0 \vee \\
& ((m\ v) = 1 \wedge \\
& (readable\ (v+1)) \wedge (A\ (v+1)) \wedge \\
& (int\ A\ m\ (m\ (v+1))) \wedge \\
& (readable\ (v+2)) \wedge (A\ (v+2)) \wedge \\
& (\tau\ A\ m\ (m\ (v+2)))))])
\end{aligned}
$$

- Theorem: *(valid intlist)*

Derived Inference Rules

$$A \equiv \lambda v.(start_read \leq v < allocptr)$$

$$\frac{m(v)=1 \quad (A\ (v+2)) \quad (int\ A\ m\ m(v+1)) \quad (intlist\ A\ m\ m(v+2))}{(intlist\ A\ m\ v)}$$

$$\frac{(intlist\ A\ m\ v) \qquad \neg(A\ w) \qquad (upd\ m\ w\ u\ m')}{(intlist\ A\ m'\ v)}$$

$$\frac{(intlist\ A\ m\ v)}{(intlist\ A'\ m\ v)}$$

$$A' \equiv \lambda v.(start_read \leq v < (allocptr + n))$$

Allocation Summary

- Using our old definition of *intlist*, we were able to prove safety of programs that traverse integer lists, but not of programs that allocate them.
- Solving this problem requires parameterizing the typing judgment by an **allocation predicate**: $(\tau\ A\ m\ v)$.
- Type definitions must satisfy certain properties about allocation: $(valid\ \tau)$.
- We can now prove safety of the program, that for example, reverses a list by allocating space for a new one and inserting appropriate values.
- We cannot handle the version that reverses pointers (mutable data structures).
- What about other data structures?

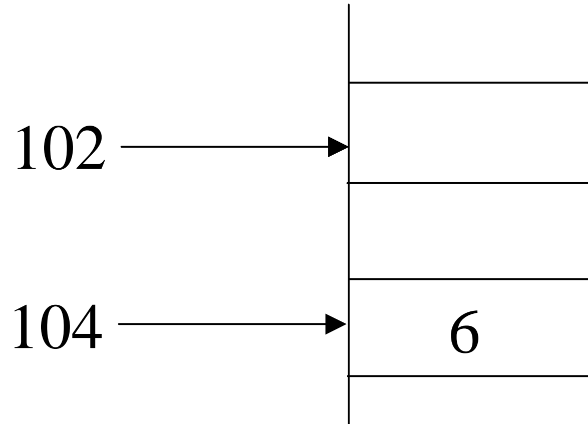
A Catalog of Type Constructors

$int \equiv \lambda A,m,v.(true)$

$constty \equiv \lambda c.\lambda A,m,v.(c = v)$

$ref \equiv \lambda \tau.\lambda A,m,v.\ ((readable\ v) \wedge (A\ v) \wedge (\tau\ A\ m\ (m\ v)))$

$offset \equiv \lambda i,\tau.\lambda A,m,v.(\tau\ A\ m\ (v+i))$



$6 :_{A,m} (constty\ 6)$

$104 :_{A,m} (ref\ (constty\ 6))$

$102 :_{A,m} (offset\ 2\ (ref\ (constty\ 6)))$

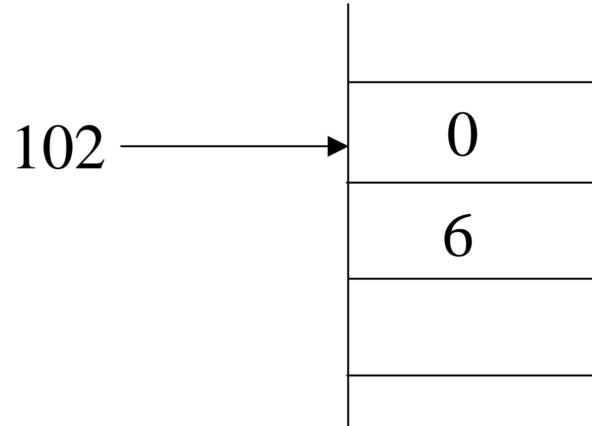
A Catalog of Type Constructors (continued)

$field \equiv i\ \tau\ \lambda A,m,v.(offset\ i\ (ref\ \tau)\ A\ m\ v)$

$intersect \equiv \lambda \tau_1, \tau_2. \lambda A,m,v.((\tau_1\ A\ m\ v) \wedge (\tau_2\ A\ m\ v))$

$union \equiv \lambda \tau_1, \tau_2. \lambda A,m,v.((\tau_1\ A\ m\ v) \vee (\tau_2\ A\ m\ v))$

$record2 \equiv \lambda \tau_1, \tau_2. \lambda A,m,v.(intersect\ (field\ 0\ \tau_1)\ (field\ 1\ \tau_2)\ A\ m\ v)$



$102 :_{A,m} (record2\ int\ (constty\ 6))$

Recursive Types in General

- Integer lists as a recursive datatype

$$intlist \equiv rec\ (\lambda\tau.\ (union\ (record1\ (constty\ 0))\ (record3\ (constty\ 1)\ int\ \tau)))$$

- Theorem: *(valid intlist)*. Proved using lemmas about validity of types and type constructors.

- Theorem: *(monotone f)* where

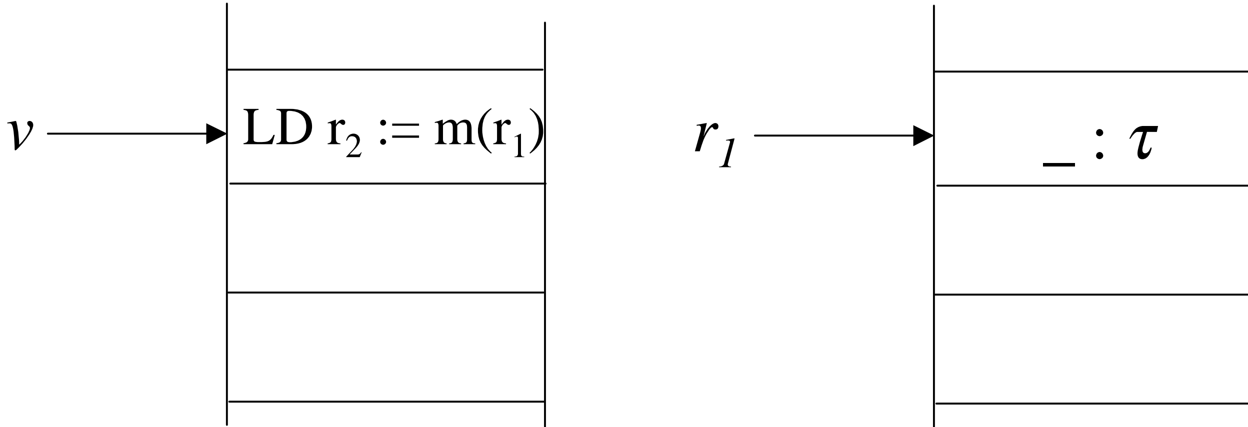
$$f \equiv rec\ (\lambda\tau.\ (union\ (record1\ (constty\ 0))\ (record3\ (constty\ 1)\ int\ \tau))).$$

  Proved using lemmas about the monotonicity of types and type constructors.

Representing Functions

$codeptr \equiv \lambda\tau.\lambda A,m,v\ldots$

…if register 1 has type τ, then it is safe to jump to address v …



Summary of Encoding Types

- We have handled lots of types (*e.g.*, most of ML).
  - Primitive types
  - User-defined datatypes, including recursive ones
  - Function types
- We haven't handled:
  - Mutable data structures
  - *All* valid ML recursive datatypes

Covariant Recursive Datatypes

- Functions as type constructors used in recursive datatypes give us covariant (but not contravariant) types.

- For example:

  $\text{intlist} = \text{nil of } (\ ) \mid \text{cons of int} \times \text{intlist}$

  $\tau_1 = c_1 \text{ of int} \mid c_2 \text{ of int} \rightarrow \tau_1$

  $\tau_2 = c_1 \text{ of int} \mid c_2 \text{ of } (\tau_2 \rightarrow \text{int}) \rightarrow \tau_2$

  $\tau_3 = c_1 \text{ of int} \mid c_2 \text{ of } ((\tau_3 \rightarrow \text{int}) \times \text{int}) \rightarrow (\tau_3 \times \text{int})$

- But not:

  $\tau_4 = c_1 \text{ of int} \mid c_2 \text{ of } \tau_4 \rightarrow \text{int}$

Contravariant Recursive Types and Functions

- Approach 1: Recursion-Theoretic Semantics as in [Mitchell & Viswanathan, ICALP'96].
  - Will allow us to handle ML, Java,…

    (We need to model mutable types also.)
  - Models types as partial equivalence relations (pers) and functions as natural numbers representing Turing machine indices.
- Approach 2: An Indexed Model of Recursive Types for Foundational Proof-Carrying Code [Appel & McAllester'00]
  - A much simpler model
  - Not as general (can't prove as many properties of programs, but perhaps not important for safety proofs)

Simplifying the Machine Semantics

- Instead of using a VCGen and proving (on paper) its soundness with respect to the abstract machine…
- We formalize the abstract machine as a definition in higher-order logic.
- We prove properties that we need (such as Hoare-like rules) as lemmas.
- See [Michael & Appel, CADE'00] for this approach applied to real machine instruction sets such as Sparc.

Informal Description of the Abstract Machine

(r,m) evaluates to:

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

Formal Description of the Abstract Machine

- Instruction decoding

$$stepRel \equiv (num \rightarrow num) \rightarrow (num \rightarrow num) \rightarrow (num \rightarrow num) \rightarrow (num \rightarrow num) \rightarrow o$$

$$decode : num \rightarrow (num \rightarrow num) \rightarrow stepRel \rightarrow o$$

$(decode\ pc\ M\ op) \equiv$

LD $r_d := r_s + c$

$$(\exists d,s,c.((M\ pc) = 2000 + d * 100 + s * 10 + c) \wedge$$
$$(op = (\lambda R,M,R',M'.M' = M \wedge$$
$$(upd\ R\ d\ (M\ ((R\ s) + c))\ R') \wedge$$
$$(readable\ ((R\ s) + c))))) \vee$$

ST : $\ldots \vee$

ADD : $\ldots \vee$ ADDC: $\ldots \vee$ BGT: $\ldots \vee$ BEQ: $\ldots$

Steps and Multisteps

- The **step** relation

$$\Rightarrow \; : \mathit{stepRel}$$

$$(R,M) \Rightarrow (R',M') \equiv$$

$$(\exists op.\ \exists R''.((\mathit{decode}\ (R\ pc)\ M\ op) \land$$
$$(\mathit{upd}\ R\ pc\ ((R\ pc)+1)\ R'') \land$$
$$(op\ R''\ M\ R'\ M')))$$

- The *multistep* rule

$$\frac{(\mathit{Inv}\ R\ M) \qquad \begin{array}{l} \forall R^1,M^1.\ (\mathit{Inv}\ R^1\ M^1) \Rightarrow \\ ((\mathit{safe}\ R^1\ M^1) \lor \\ \exists R^2,M^2.\ (((R^1,M^1) \Rightarrow (R^2,M^2)) \land (\mathit{Inv}\ R^2\ M^2))) \end{array}}{(\mathit{safe}\ R\ M)}$$

Initial State and Exit

- Interface rules, *e.g.*,
  - Exiting by jumping to a designated address is *safe*.
  - The program counter is initially set to the first instruction of the code.

$$\frac{(R\ pc) = return_addr}{(safe\ R\ M)} \qquad (R^0\ pc) = start_code$$

- By proving these rules from the machine semantics, we essentially formalize Necula's proof of soundness of VCGen.

A Prototype Implementation

- Like PCC, Foundational PCC has been implemented in the Twelf system which implements the Logical Framework (LF or λP).
  - A first prototype was implemented in λProlog
  - We have seen a λterm notation for λHOL proofs. Here, we have a shallow embedding in LF.
  - We experimented with many versions of the logic (safety policy). Using a logical framework allowed us to change it easily.

Other Results

- Michael & Appel, CADE 2000 show how to encode the semantics of real machine architectures such as Sparc and Mips.
- Not all ML datatypes are fit our definition of monotone. An indexed model of recursive types handles a larger class (Appel & McAllester 2000).
- Ahmed, Appel, & Virga, LICS 2002 show how to add mutable datatypes to the indexed model.
- Swadi, Appel, & Virga, 2001 presents a typed machine language to which high-level languages may be compiled.
- Necula, CADE 2002 formalizes a proof of soundness of typing rules.
- Appel & Felty have further developed an environment for implementing proof-carrying code systems (papers to appear in TPLP and JFP).
- Shao et. al. have developed a syntactic approach to PCC, LICS 2002.
- Bernard & Lee, Temporal Logic for Proof-Carrying Code, CADE 2002.

Other Ongoing and Future Work

- An Environment for Proof-Carrying Code
- Automating Proofs of Safety
- Modeling Semantics of FLINT Types
- Handling Other Programming Languages such as Java
- Proof Size
- Mutable Fields
- Certified Compilation
- Machine Instruction Sets such as Sparc and Pentium
- Concurrency
- Runtime Code Generation
- Garbage Collection

Some Other Approaches to Software Safety

- **Sandboxing** inserts extra instructions to bound the range of accessible addresses [Wahbe et.al.'93]
- In **Java bytecode verification**, the just-in-time compiler is in the Trusted Code Base
- **Policy-Directed Code Safety** [Evans & Twyman'99] provides a system architecture for expressing safety policies, which are enforced by transforming programs.
- **Typed Assembly Language** [Morrisett et.al.] extends traditional untyped assembly languages with typing annotations, memory management primitives, and a sound set of typing rules.
- **Certified Binaries** [Shao, Saha, Trifonov, & Papaspyrou, POPL'02] integrate an entire proof system (the calculus of inductive constructions) into a compiler intermediate language.