

# A Compositional Logic for Control Flow

Gang Tan

Boston College

October 19, 2005 at Church Seminar, Boston University

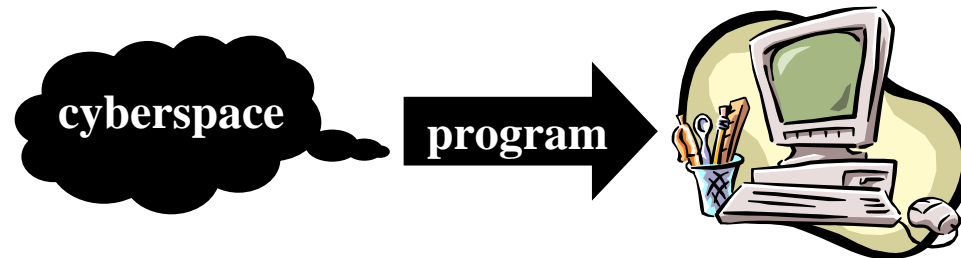
My thesis work with Prof. Andrew W. Appel  
at Princeton University

# Talk Outline

- ➔ Overview of Foundational Proof-Carrying Code (FPCC)
- Why we need a new program logic for machine-language programs?
- $\mathcal{L}_c$ : A Logic for Machine-Language Programs
- Formal Semantics of  $\mathcal{L}_c$
- $\mathcal{L}_c$  in FPCC

# Mobile Code Security

- Protect trusted system against untrusted code



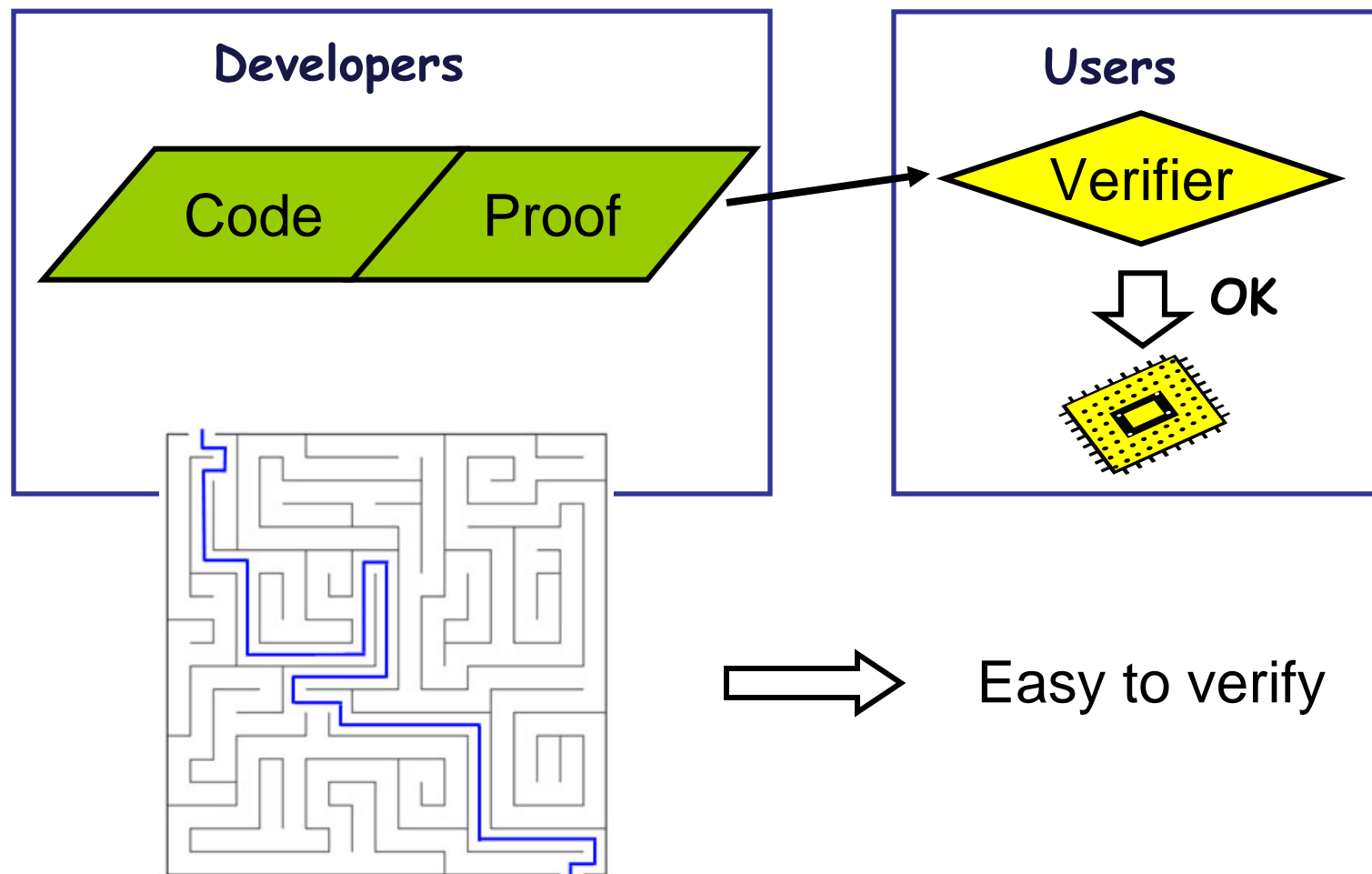
- Everybody loves extensibility
  - Extensible OS kernel
  - Web browsers, routers, switches, ...
  - Even PowerPoint has VB add-ins

How to give foreign code direct access without compromising host integrity?

# Proof-Carrying Code (PCC), in a Nutshell

[Necula & Lee 97]

- Code + Proof



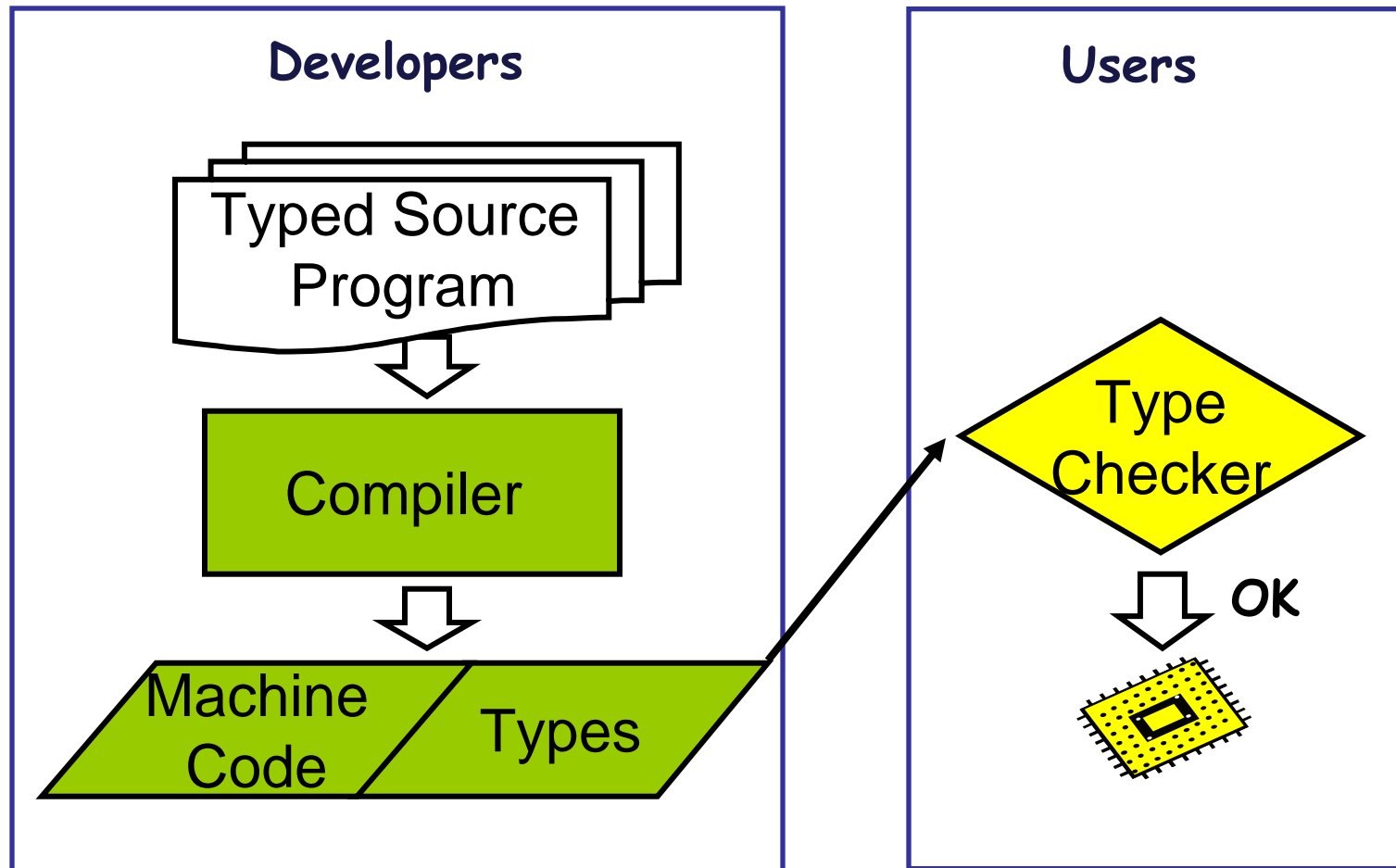
# Proof-Carrying Code (PCC)

- Focus on type safety
  - Implies many other important properties
    - No buffer overflow; control-flow safety; ...
  - Safety proofs can be generated automatically

# Proof-Carrying Code (PCC), Typed-Assembly Language (TAL)

[Necula & Lee 97]

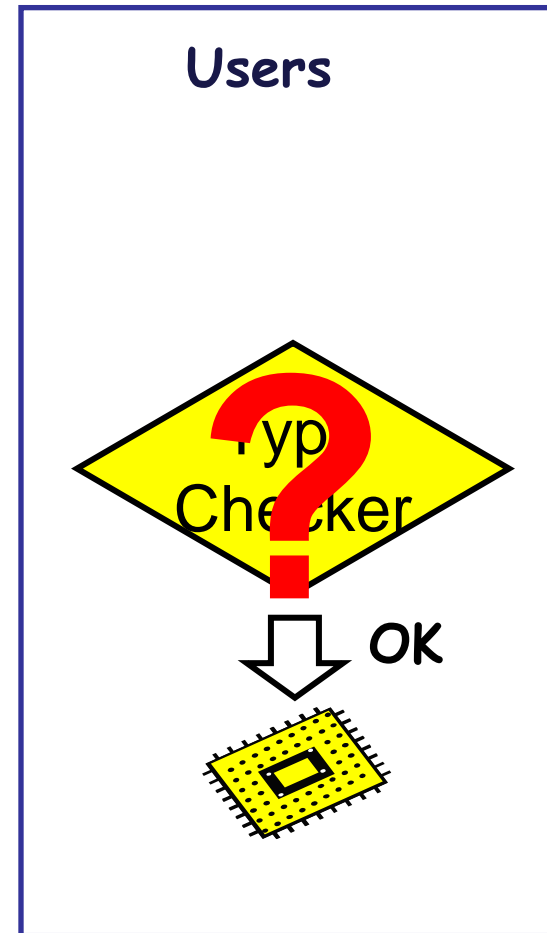
[Morrisett et al. 98]



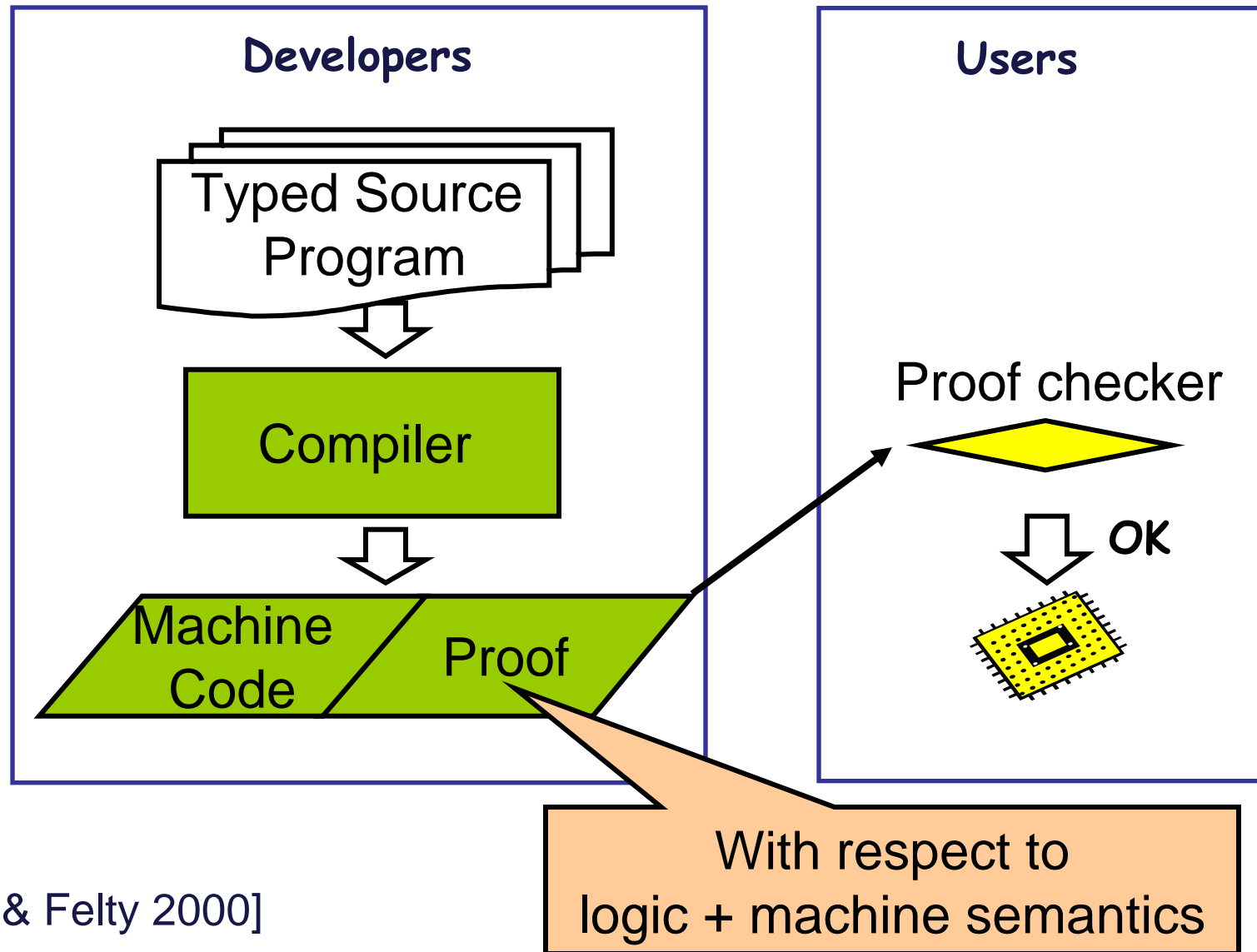
No need to trust code developers  
No need to trust the compiler

# Can We Trust the Type System?

- Production-scale low-level type systems
  - Huge: LTAL has 1200 operators & rules!
  - Complex: because of intricate machine semantics
  - We routinely found and fixed bugs in early versions of LTAL

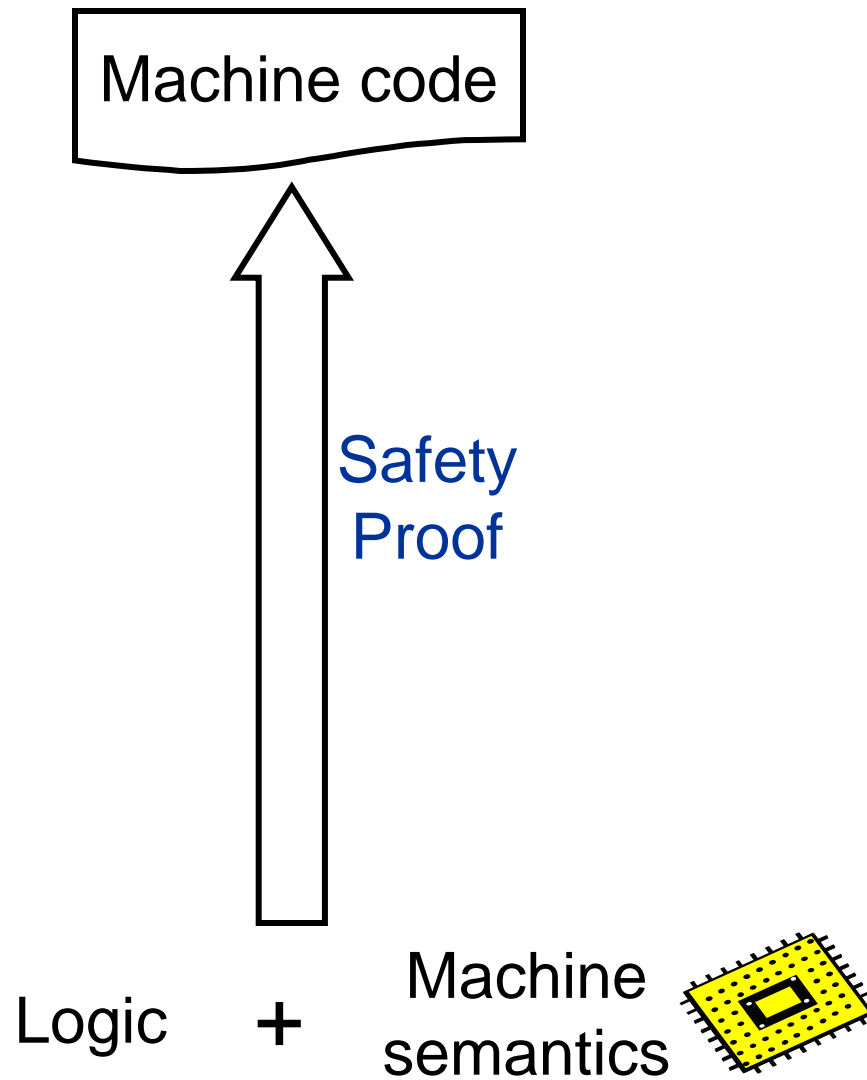


# Foundational Proof-Carrying Code (FPCC)

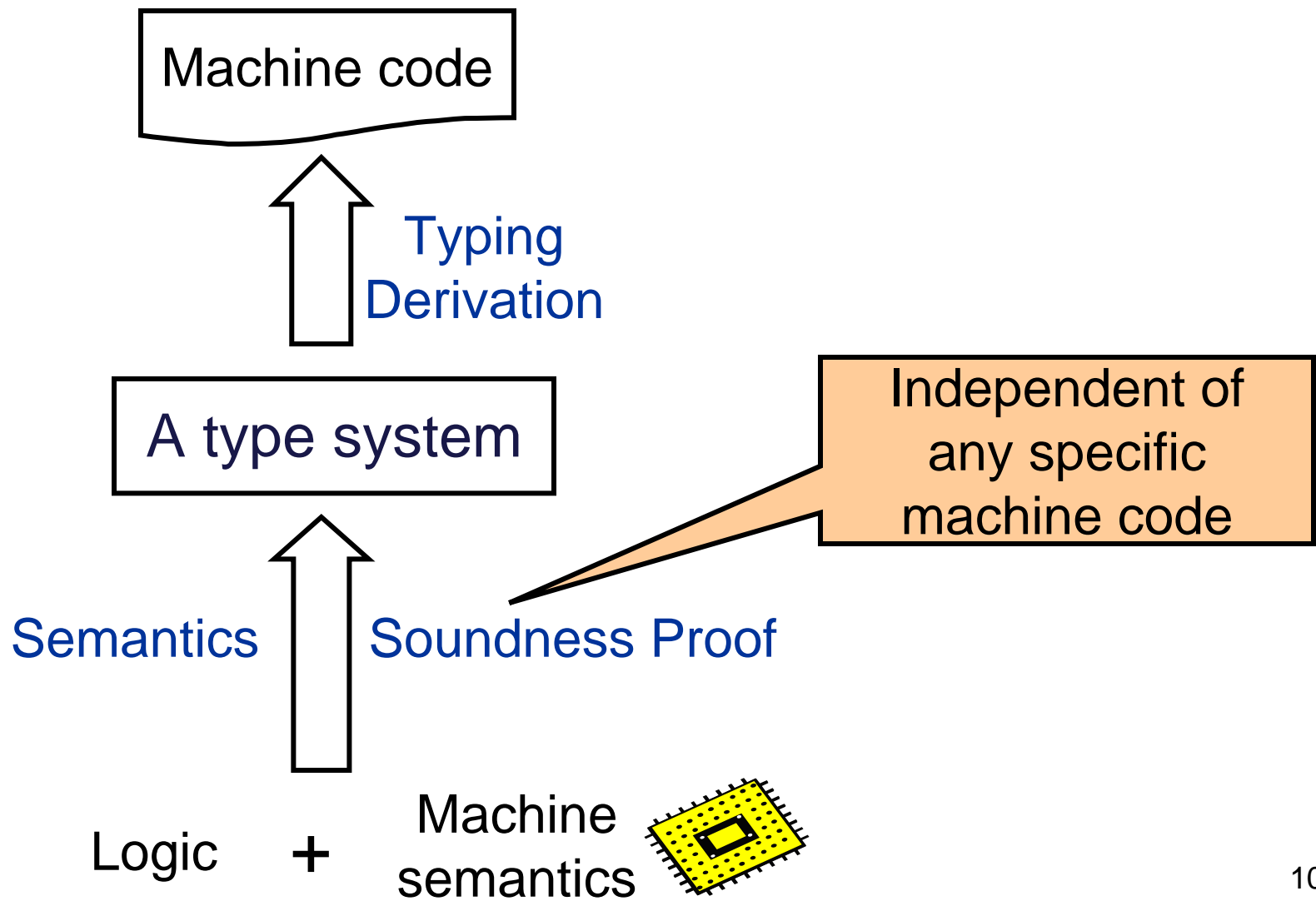


[Appel & Felty 2000]

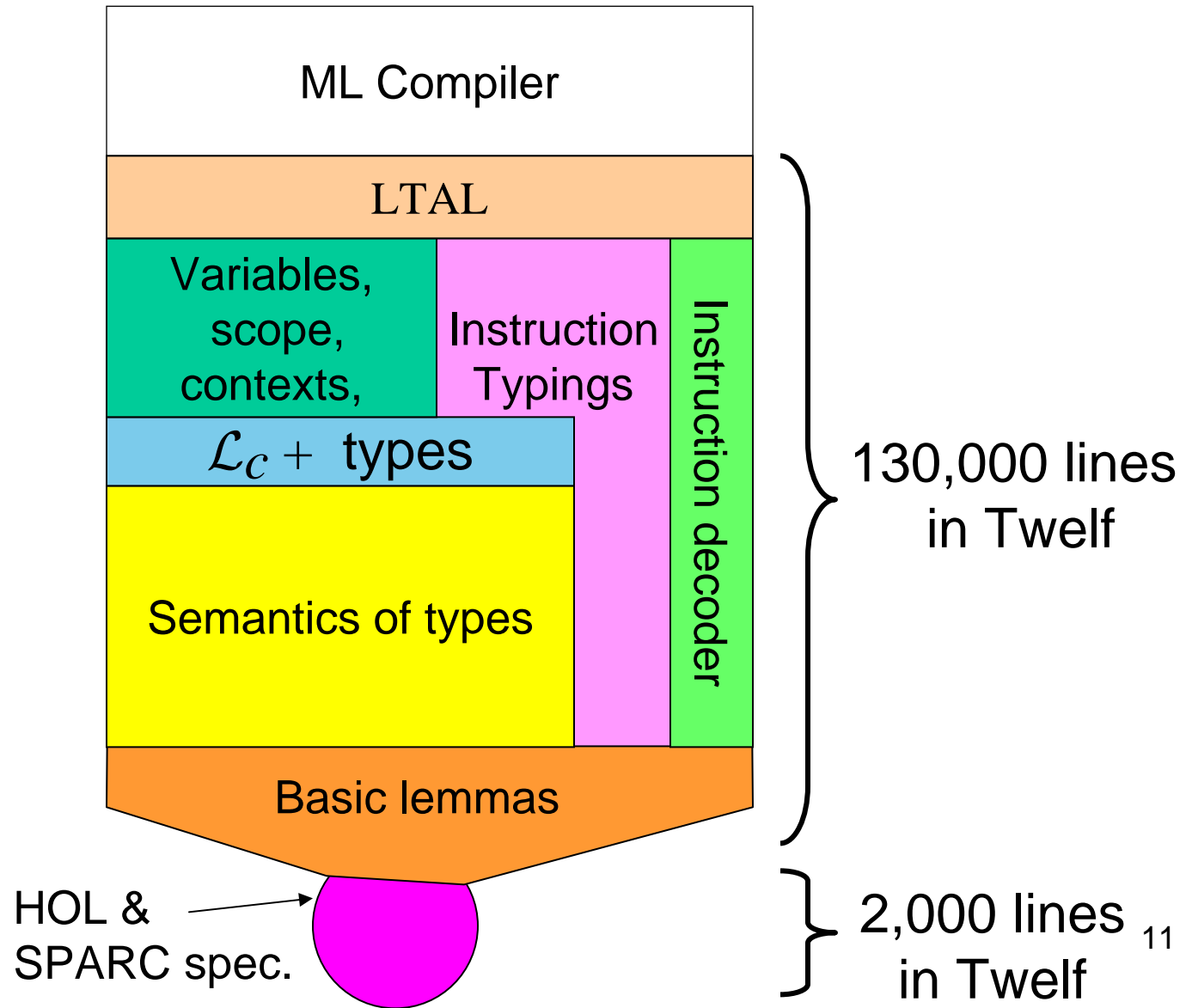
# Use Type Systems to Organize the Big Proof



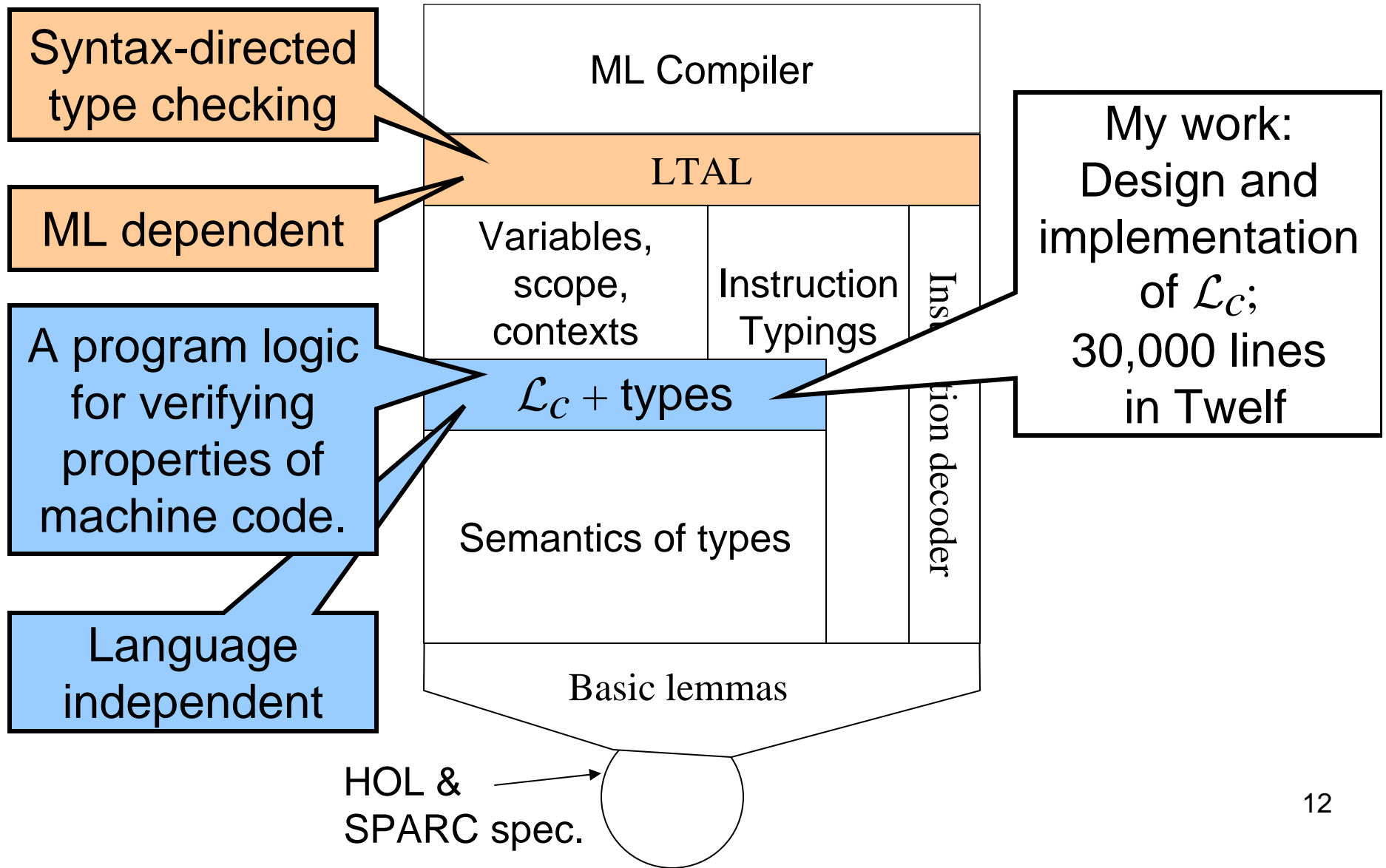
# Use Type Systems to Organize the Big Proof



# FPCC Project Overview



# FPCC Project: A Two-layer Design

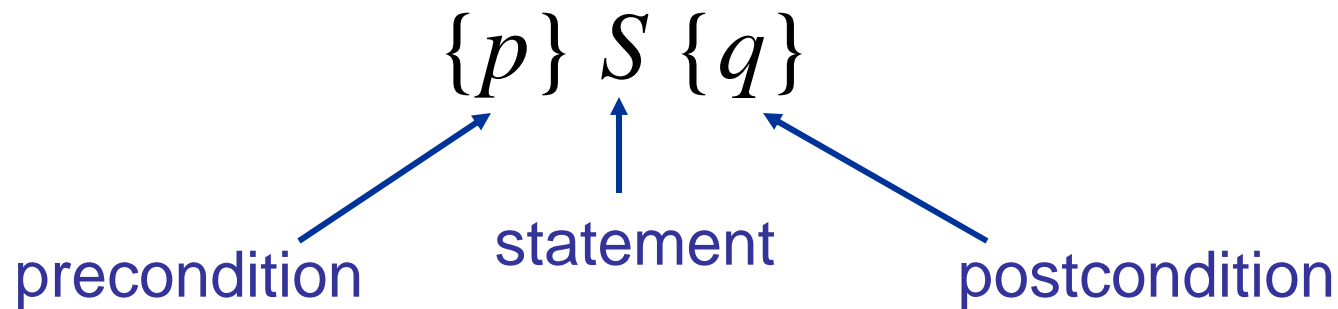


# Talk Outline

- Overview of Foundational Proof-Carrying Code (FPCC)
- Why we need a new program logic for machine-language programs?
- $\mathcal{L}_c$ : A Logic for Machine-Language Programs
- Formal Semantics of  $\mathcal{L}_c$
- $\mathcal{L}_c$  in FPCC

# Background: Hoare Logic

- Specification using **Hoare triple**:

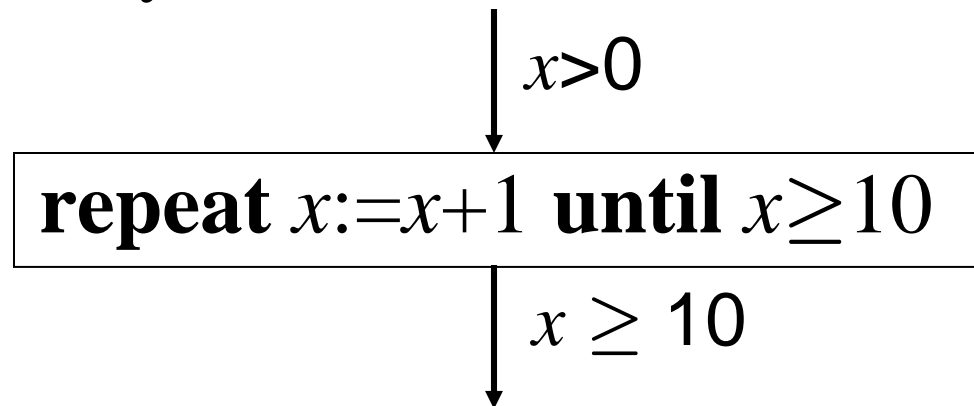


An example:

$\{x > 0\}$  **repeat**  $x := x + 1$  **until**  $x \geq 10$   $\{x \geq 10\}$

# Hoare Logic is for Structured Programs

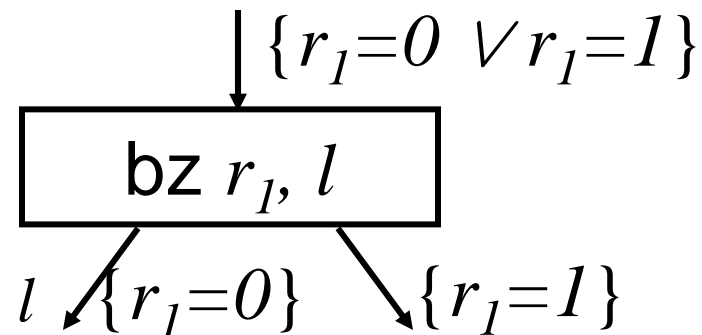
- Structured programs: no gotos
  - Written using constructs such as “if-then-else”, “repeat-until”, “while-do”, ...
  - Each program fragment has exactly **one** entry and **one** exit.



- Hoare triple:  $\{p\} S \{q\}$

# Hoare Logic: not Suitable for Machine-Language Programs

- Unstructured programs
  - Goto statements with unrestricted destinations.
  - Each program fragment has possibly **multiple** entries and **multiple** exits.



*... [in Hoare Logic], it is not surprising that trouble arises in considering program segments with more than one mode of entry and/or exit. -- Michael J. O'Donnell, 1982*

# Previous Work: Program Logics for Unstructured Programs

- Early work
  - Clint & Hoare 69; Kowaltowski 77; Arbib & Alagic 79; de Bruin 81; TAL: Morrisett et al. 98
- de Bruin's system
  - Separate rules for different control-flow constructs
  - Not modular: Need global invariants

# de Bruin's System: Need Global Invariants

$$\langle \underline{L_1 : p_1, \dots, L_n : p_n} \mid \{p\} s \{q\} \rangle$$

**Global** invariant:  
all **label** invariants in a program

- Composition requires matching of global labels

$$\left. \begin{array}{l} \square \langle \quad \mid \{x>0\} x := x + 1 \quad \{x>0\} \rangle \\ \square \langle l:(x>0) \mid \{x>0\} \mathbf{if} x<10 \mathbf{goto} l \{x \geq 10\} \rangle \end{array} \right\} \text{☹: Cannot Compose!}$$

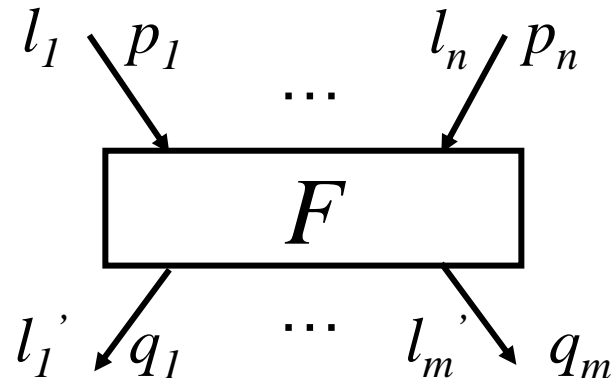
# Talk Outline

- Overview of Foundational Proof-Carrying Code (FPCC)
- Why we need a new program logic for machine-language programs?
- $\mathcal{L}_c$ : A Logic for Machine-Language Programs
- Formal Semantics of  $\mathcal{L}_c$
- $\mathcal{L}_c$  in FPCC

# Multiple Entries and Multiple Exits

- Reasoning units: Multiple-entry and multiple-exit program fragments

Informal syntax:



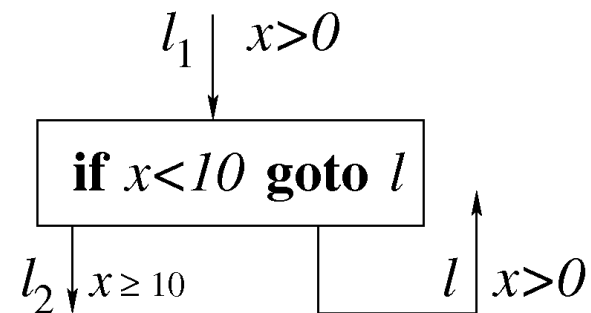
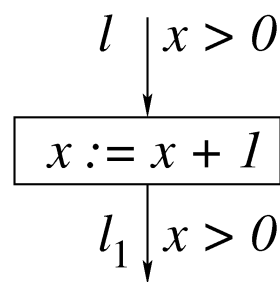
Formal syntax:

$$F ; \underbrace{\{l'_1 \triangleright q_1, \dots, l'_m \triangleright q_m\}}_{\text{Exits, } \Phi} \vdash \underbrace{\{l_1 \triangleright p_1, \dots, l_n \triangleright p_n\}}_{\text{Entries, } \Psi}$$

# Rules in $\mathcal{L}_c$

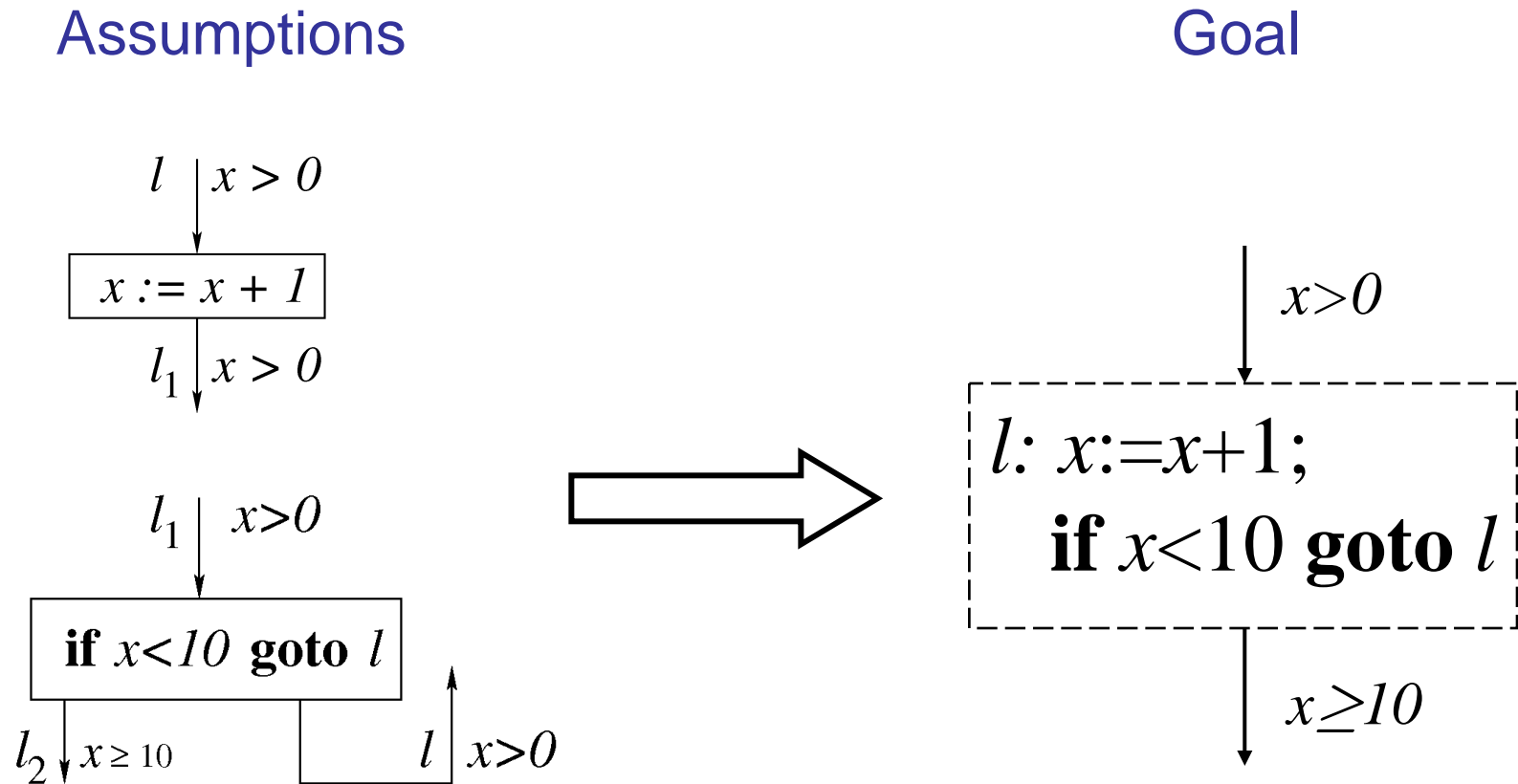
- $\mathcal{L}_c$ : a formal system
  - Rules for individual statements
  - Rules for composing statements
- Rules for individual statements

Examples:

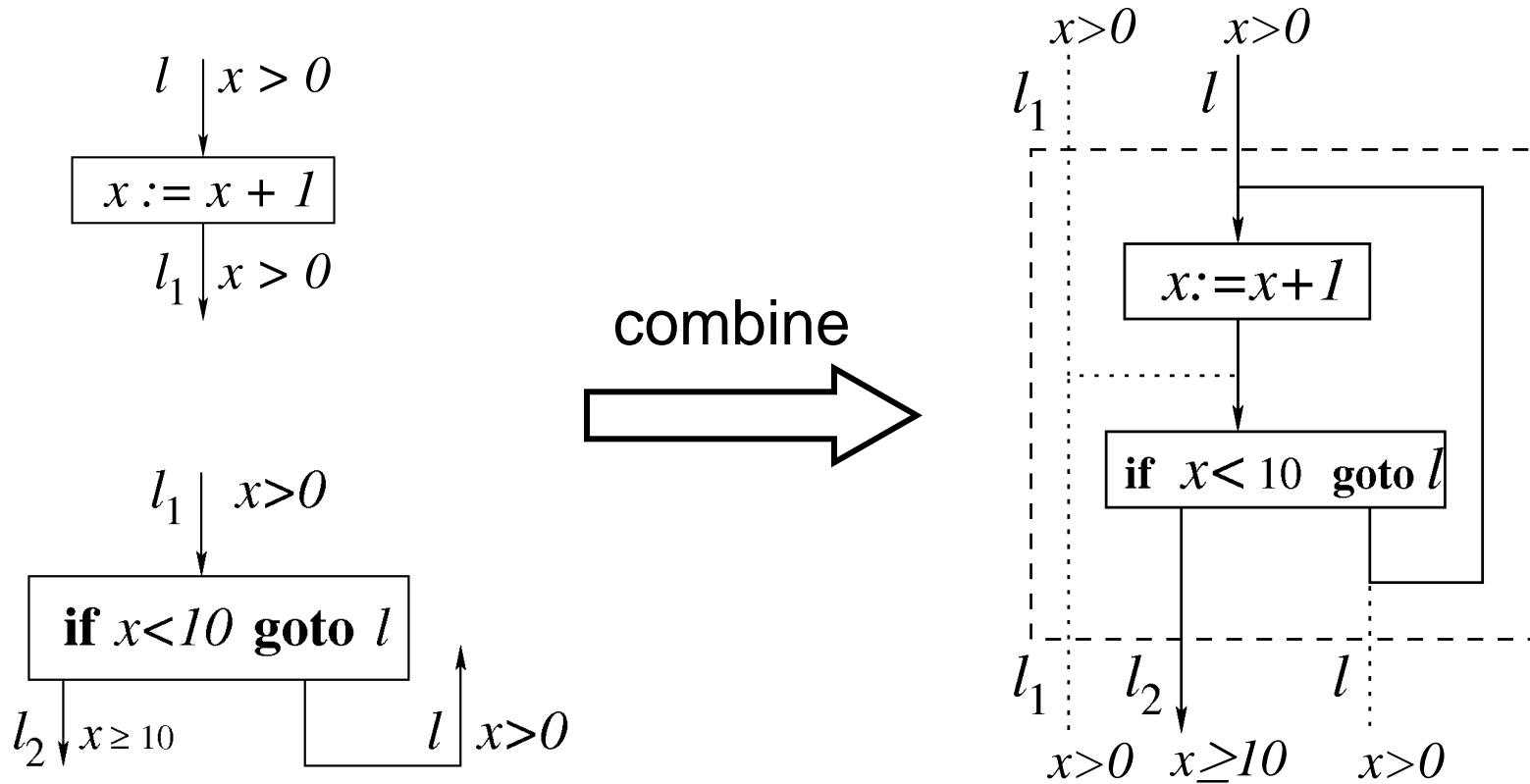


# Composition Rules

- Compose fragments together to form properties on the combined fragment

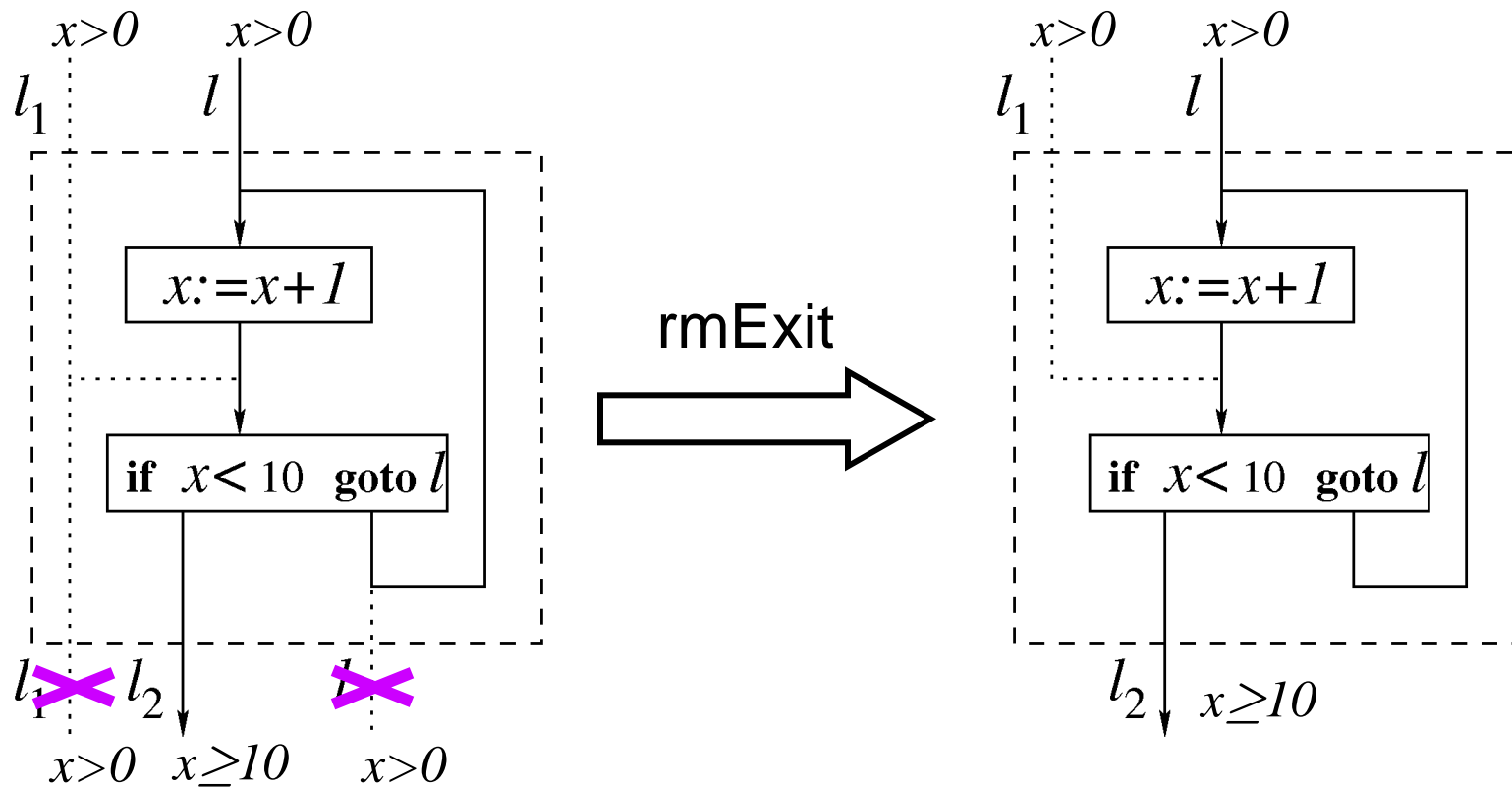


# Step 1: Combining Fragments



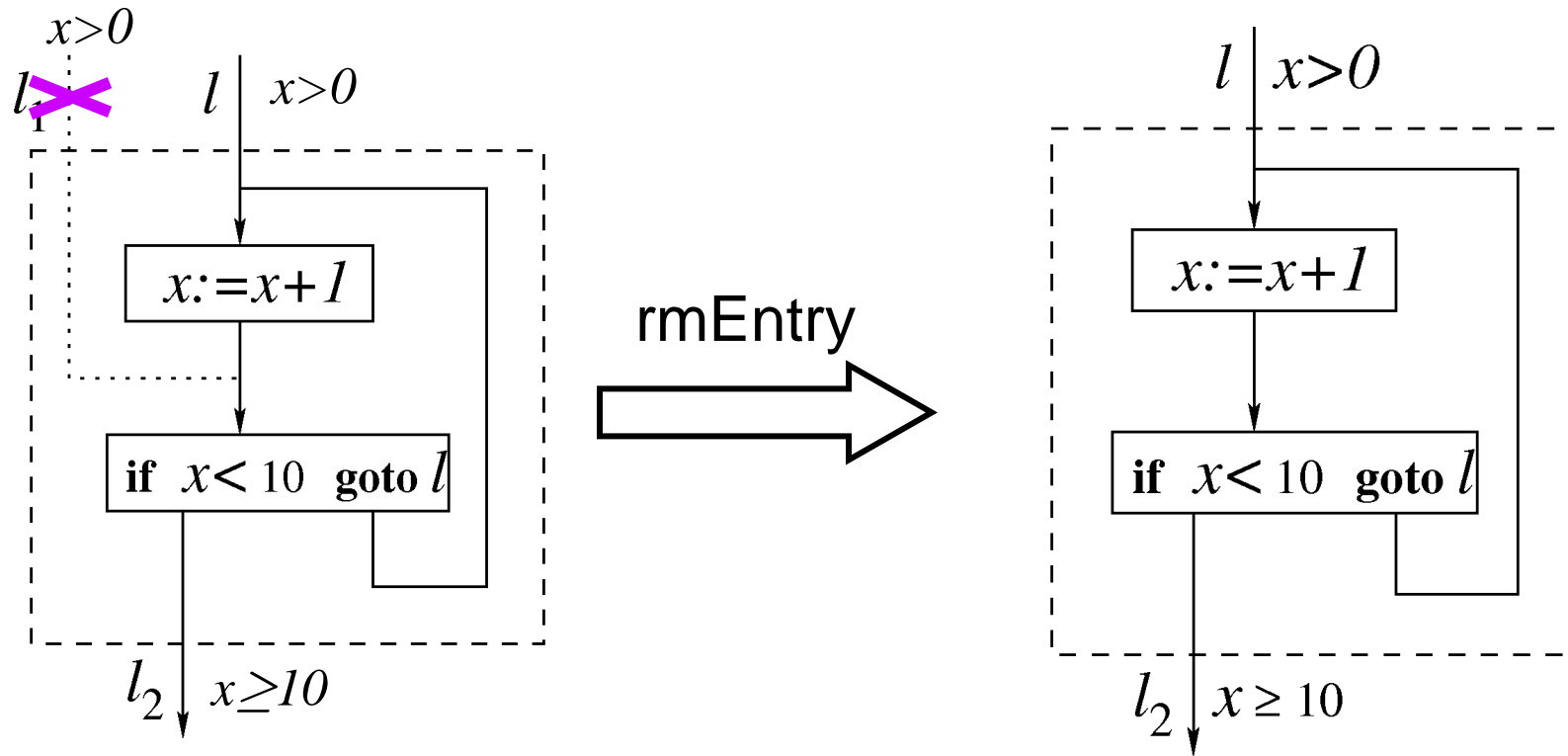
$$\frac{F_1; \Phi_1 \vdash \Psi_1 \quad F_2; \Phi_2 \vdash \Psi_2}{F_1 \cup F_2; \Phi_1 \cup \Phi_2 \vdash \Psi_1 \cup \Psi_2} \text{combine}$$

# Step 2: Removing Exits



$$\frac{F ; \Phi \cup \{l \triangleright p\} \vdash \Psi \cup \{l \triangleright p\}}{F ; \Phi \vdash \Psi \cup \{l \triangleright p\}} \text{rmExit}$$

# Step 3: Removing Entries



$$\frac{F; \Phi \vdash \Psi_1 \cup \Psi_2}{F; \Phi \vdash \Psi_1} \text{rmEntry}$$

# $\mathcal{L}_c$ 's Composition Rules

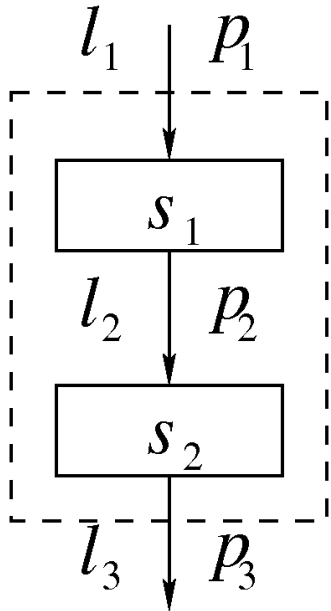
$$\frac{F_1; \Phi_1 \vdash \Psi_1 \quad F_2; \Phi_2 \vdash \Psi_2}{F_1 \cup F_2; \Phi_1 \cup \Phi_2 \vdash \Psi_1 \cup \Psi_2} \text{combine}$$

$$\frac{F; \Phi \cup \{l \triangleright p\} \vdash \Psi \cup \{l \triangleright p\}}{F; \Phi \vdash \Psi \cup \{l \triangleright p\}} \text{rmExit}$$

$$\frac{F; \Phi \vdash \Psi_1 \cup \Psi_2}{F; \Phi \vdash \Psi_1} \text{rmEntry}$$

- Fine-grained composition rules
  - Support reasoning about unstructured control flow
  - Support derivation of rules for common control-flow structures

# Derivation of Sequential Composition



$$\begin{array}{l} \{l_1 : (s_1) : l_2\}; \{l_2 \triangleright p_2\} \vdash \{l_1 \triangleright p_1\} \\ \{l_2 : (s_2) : l_3\}; \{l_3 \triangleright p_3\} \vdash \{l_2 \triangleright p_2\} \end{array}$$

$$\frac{\quad}{\{l_1 : (s_1; s_2) : l_3\}; \{\cancel{l_2 \triangleright p_2}, l_3 \triangleright p_3\} \vdash \{l_1 \triangleright p_1, l_2 \triangleright p_2\}} \text{combine}$$

$$\frac{\quad}{\{l_1 : (s_1; s_2) : l_3\}; \{l_3 \triangleright p_3\} \vdash \{l_1 \triangleright p_1, \cancel{l_2 \triangleright p_2}\}} \text{rmExit}$$

$$\frac{\quad}{\{l_1 : (s_1; s_2) : l_3\}; \{l_3 \triangleright p_3\} \vdash \{l_1 \triangleright p_1\}} \text{rmEntry}$$

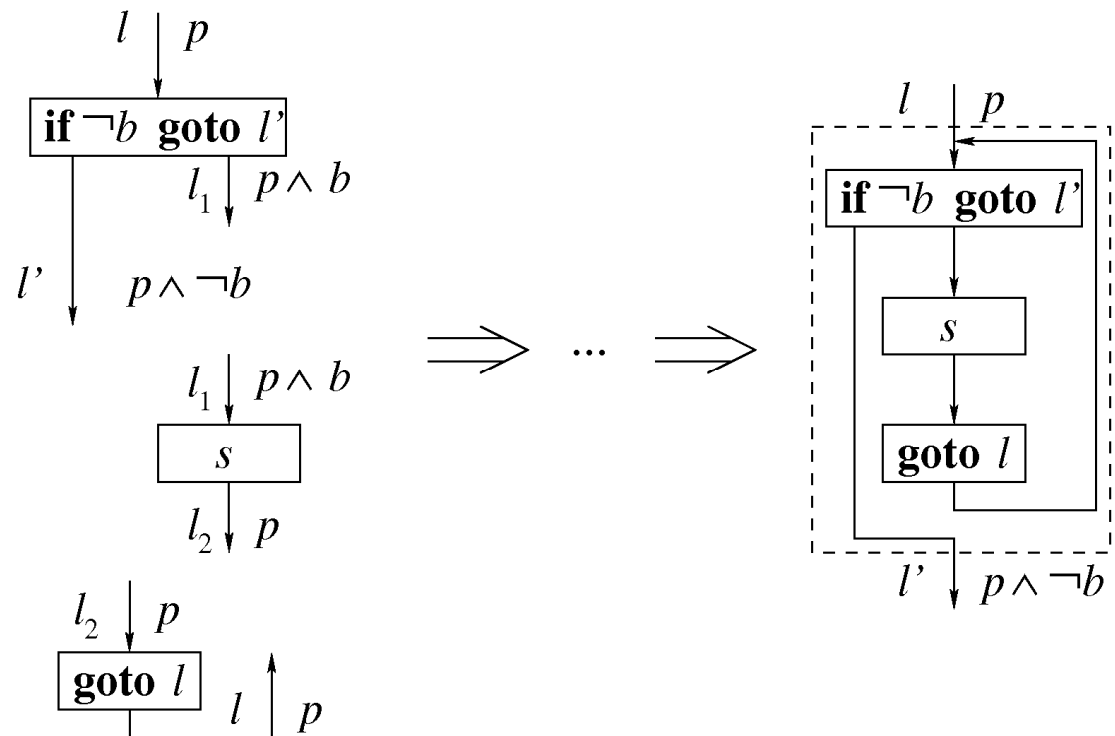
# Deriving The Rule for While Loops

In Hoare Logic:

$$\frac{\{p \wedge b\} s \{p\}}{\{p\} \text{while } b \text{ do } s \{p \wedge \neg b\}}$$

**while  $b$  do  $s$  :**

$l : \text{if } \neg b \text{ goto } l'$   
 $l_1 : s$   
 $l_2 : \text{goto } l$   
 $l' :$



# Talk Outline

- Overview of Foundational Proof-Carrying Code (FPCC)
- Why we need a new program logic for machine-language programs?
- $\mathcal{L}_C$ : A Logic for Machine-Language Programs
  - ➔ Formal Semantics of  $\mathcal{L}_C$ 
    - ➔ Give  $\mathcal{L}_C$  a semantics in higher-order logic
    - ➔ Fit into FPCC
    - ➔ A naïve semantics for  $F$ ;  $\Phi \vdash \Psi$  won't work
- $\mathcal{L}_C$  in FPCC

# Machine States and Step relation

- A machine state  $\sigma$
- Small step operational semantics
  - $\sigma \mapsto \sigma'$
- Multiple steps:  $\sigma \mapsto^* \sigma'$

# Semantics of $l \triangleright p$ : Continuations

- $l \triangleright p$  being true in a state  $\sigma$

Safe to continue from the label  $l$   
provided that the precondition  $p$  is met

$$\begin{aligned} \sigma &\models l \triangleright p \\ &\triangleq (\text{control}(\sigma) = l) \wedge (p \text{ true in } \sigma) \\ &\Rightarrow (\sigma \mapsto^* \dots) \end{aligned}$$

- $l \triangleright p$  being **approximately** true:

$$\begin{aligned} \sigma &\models_k l \triangleright p \\ &\triangleq (\text{control}(\sigma) = l) \wedge (p \text{ true in } \sigma) \\ &\Rightarrow (\sigma \mapsto^k \dots) \end{aligned}$$

# Semantics of $F; \Phi \vdash \Psi$

- A set of continuations being approx. true

$$\begin{aligned} & \sigma \models_k \Psi \\ \triangleq & \forall (l \triangleright p) \in \Psi. \sigma \models_k l \triangleright p \end{aligned}$$

- Semantics:

$$\begin{aligned} & F; \Phi \models \Psi \\ \triangleq & \forall \sigma, k. \text{loaded}(F, \sigma) \wedge \sigma \models_k \Phi \\ & \Rightarrow \sigma \models_{k+1} \Psi \end{aligned}$$

Requires at least one computation step  
from an entry to an exit

# Why is the one-step requirement?

- Because of the rmExit rule:

$$\frac{F; \Phi \cup \{l \triangleright p\} \vdash \Psi \cup \{l \triangleright p\}}{F; \Phi \vdash \Psi \cup \{l \triangleright p\}} \text{rmExit}$$

When both  $\Phi$  and  $\Psi$  are empty:

$$\frac{F; \{l \triangleright p\} \vdash \{l \triangleright p\}}{F; \{\} \vdash \{l \triangleright p\}}$$

Without the one-step requirement, the rule is like:  
From “A imply A”, derive A.

Our semantics assume the left to index  $k$ , and  
prove the right to index  $k+1$

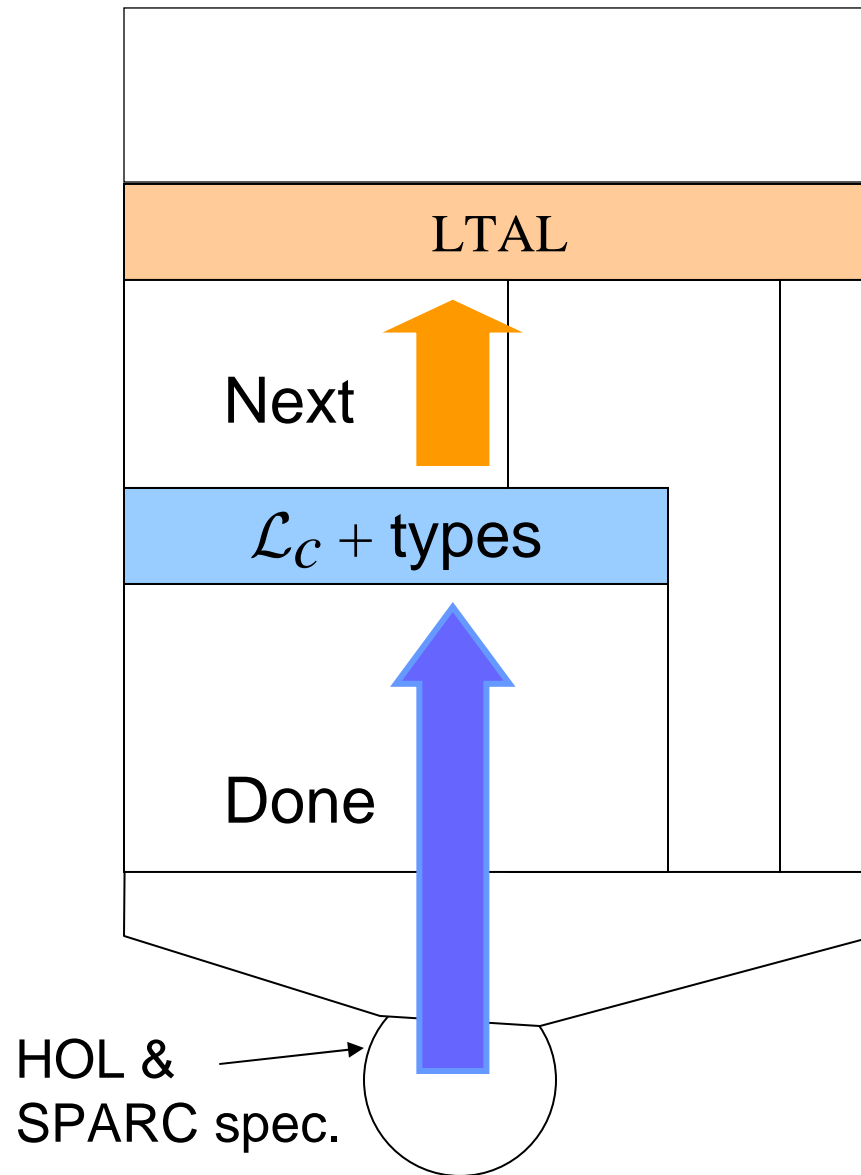
# Soundness and Completeness

- **Soundness:** *If  $F; \Phi \vdash \Psi$ , then  $F; \Phi \models \Psi$ .*
  - By induction over the derivation of  $F; \Phi \vdash \Psi$
- **Completeness:** *If  $F; \Phi \models \Psi$ , then  $F; \Phi \vdash \Psi$ .*
  - With some assumptions:
    - Assume a complete derivation system for the assertion language
    - Assume the assertion language is expressive enough
  - Adaptation of Cook's completeness proof for Hoare Logic

# Talk Outline

- Overview of Foundational Proof-Carrying Code (FPCC)
- Why we need a new program logic for machine-language programs?
- $\mathcal{L}_c$ : A Logic for Machine-Language Programs
- Formal Semantics of  $\mathcal{L}_c$
- $\mathcal{L}_c$  in FPCC

# From $\mathcal{L}_c$ to LTAL



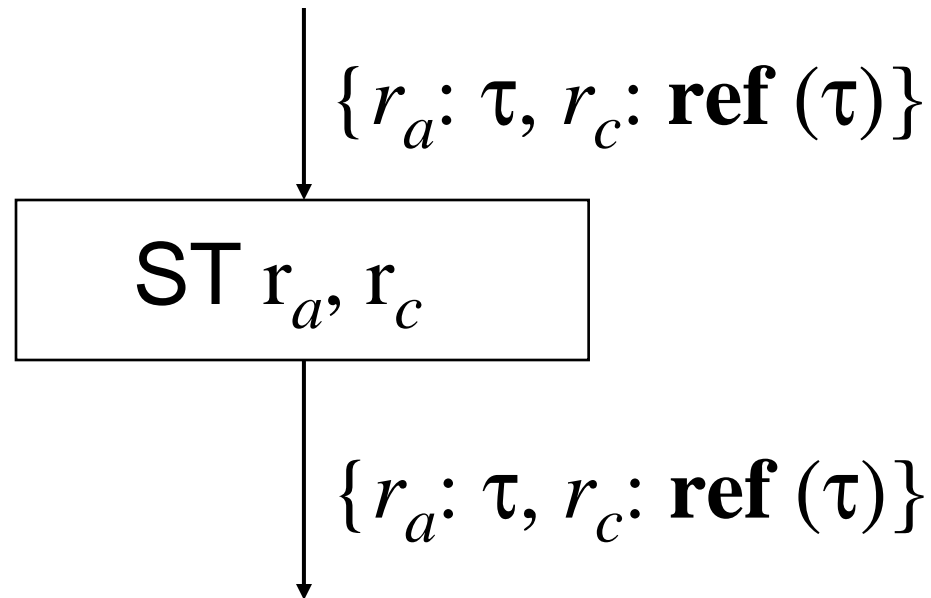
# From $\mathcal{L}_c$ to LTAL

- Need to accommodate LTAL's features
  - Polymorphic and existential types
  - Virtual instructions
  - Indirect jumps and pc-relative jumps
  - Memory initialization/allocation


# Hardwork:


## Many Instruction Rules in LTAL


- Memory store instruction:  $\text{ST } r_a, r_c$
- Rule for type-preserving updating store:




# In Twelf Syntax

```
jdginstr_i_ST_update_st :  Name of the rule
  pf (tml_valid @ L) -> pf (tml_valid @ Tau)
  -> pf (tml_haskind @ L @ kd_numeric)
  -> pf (tml_haskind @ Tau @ kd_scalar)
  -> pf (tml_valid @ Phi)
  -> pf (sub_tml @ Phi @ (A hasty Tau))
  -> pf (reg_imm_hasty @ Phi @ Bx @ (const_ty @ SN))
  -> pf (sub_tml @ Phi
        @ (C hasty (offset_ty @ (const_ty @ SN)
                       @ (ref_ty @ Tau))))
  -> pf (realreg @ A @ Ar) -> pf (imode_or_rmode @ Bx)
  -> pf (realreg @ C @ Cr)
  -> pf (decode (ST Ar (inj_generalA Cr Bx)) W)
  -> pf (jdginstr @ ((nl_singleton W) nl_@ L)
        @ ((plus32_n_tml L four) cont Phi)
        @ (L cont Phi)) =
  ... ..
  ... ..
```

 Assumptions

 Conclusion

 Proofs

# Machine-Specific Issues: A lot of Engineering

```
jdginstr_i_ST_update_st :
```

```
...
```

```
-> pf (sub_tml @ Phi  
      @ (C hasty (offset_ty @ (const_ty @ SN)  
                @ (ref_ty @ Tau))))
```

```
...
```

```
-> pf (imode_or_rmode @ Bx)  
-> pf (decode (ST Ar (inj_generalA Cr Bx)) W)  
-> pf (jdginstr @ ((nl_singleton W) nl_@ L)  
      @ ((plus32_n_tml L four) cont Phi)  
      @ (L cont Phi)).
```

# A Library of Lemmas about SPARC Instructions

- Arithmetic instructions
  - add, sub, smul
- Branching instructions
  - ba (branch always)
  - jmpl (jump and link)
  - bz, bl, ... (conditional branches)
- Set condition codes
  - subcc
- Memory instructions
  - ld, st

Proved only those that are used by our compiler

# Additional Related Work

- Appel & McAllester 2001
  - Indexed model of types
  - All types are indexed by computational steps
- Benton 2005, Saabas & Usstalu 2005
  - Labels are associated with pre and post conditions