

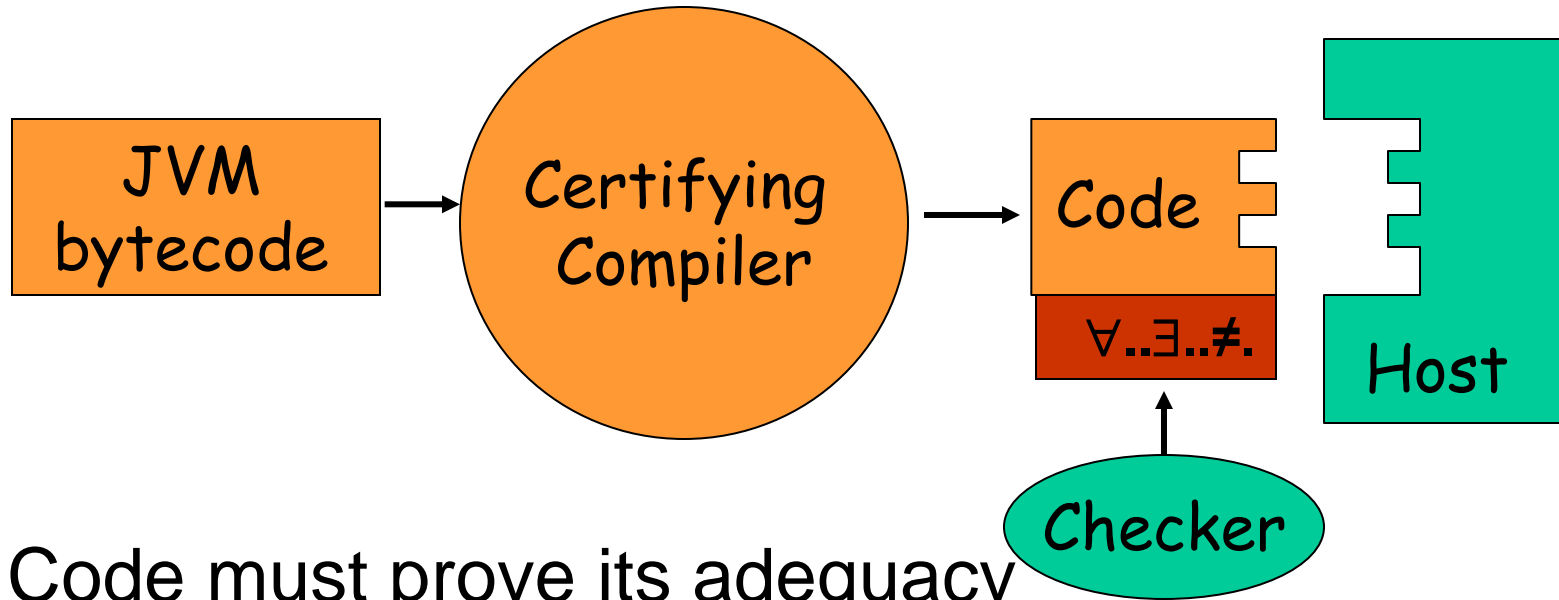
Typed Assembly Language (TAL)

Lujo Bauer

18-732

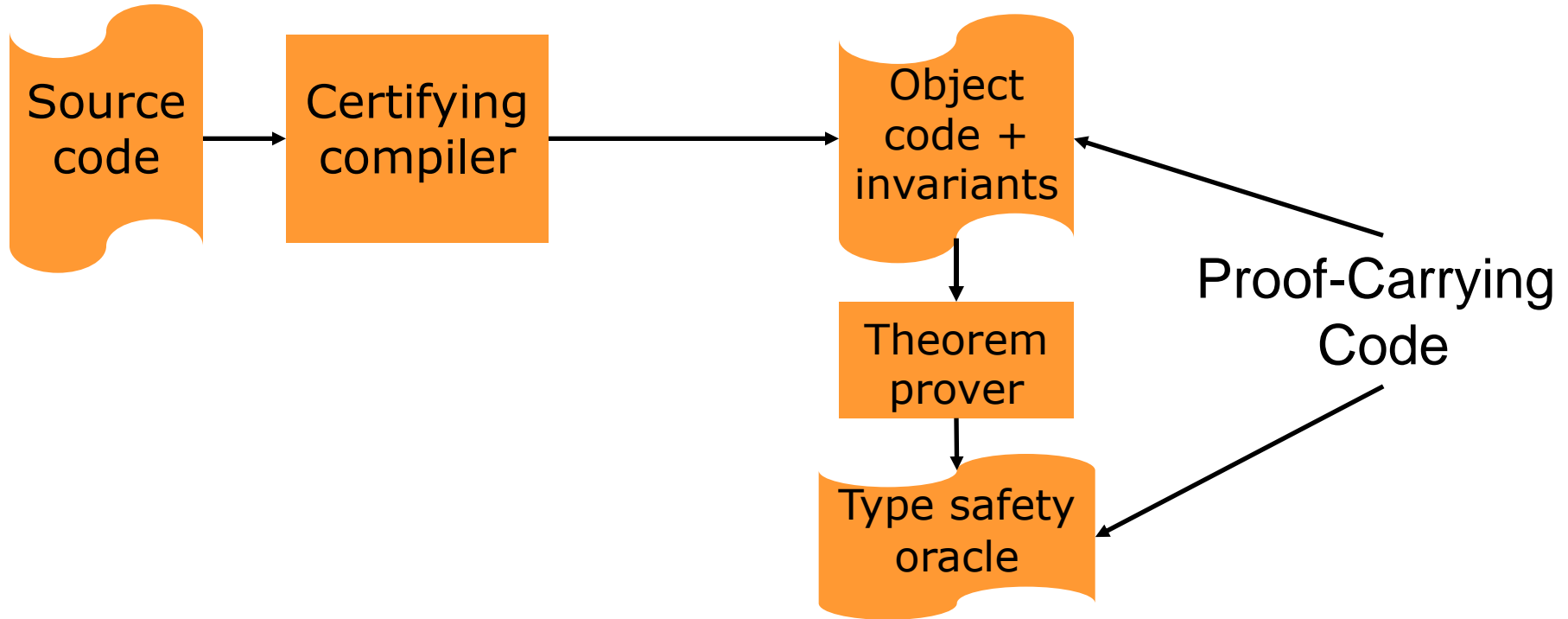
Fall 2010

Assurance Support: Proof-Carrying Code (PCC)



- Code must prove its adequacy
- Hard to prove but easy to check
- Works even for machine code—check what you run
- One (simple) checker for many policies
- Trust very little

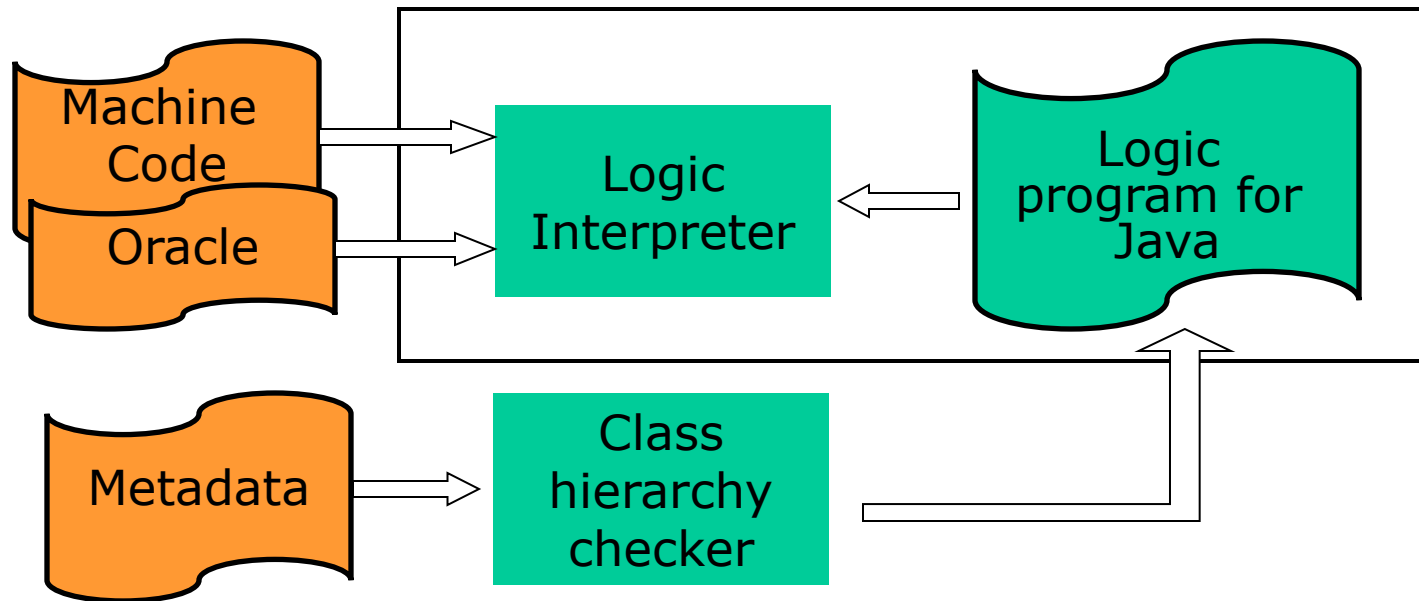
PCC via Certifying Compilation



- Implementation:
 - The compiler emits typing invariants for registers
 - The prover reconstructs the proof of type safety

Instantiating PCC to Java

- The PCC verifier for Java:



- Replace the bytecode verifier with PCC checker
- The metadata checker adds new logic rules to describe the class hierarchy

What Is TAL?

A type system for assembly language(s):

- built-in abstractions (tuple, code)
- operators to build new abstractions
- annotations on assembly code
- an abstraction checker

Many theorems carry over to assembly code

What Is TAL?

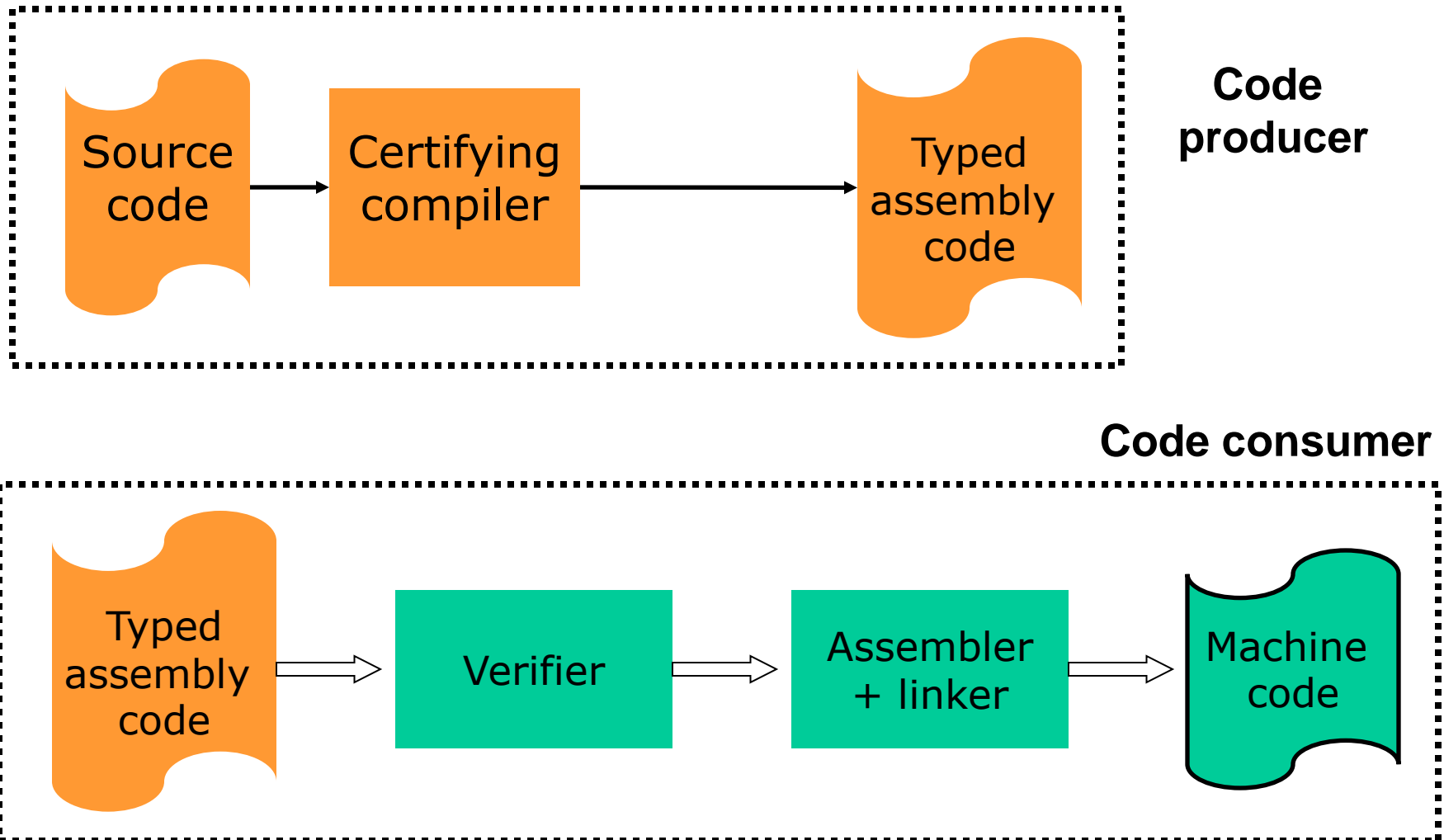
Theory:

- small RISC-style assembly language
- compiler from System F to TAL
- soundness and preservation theorems

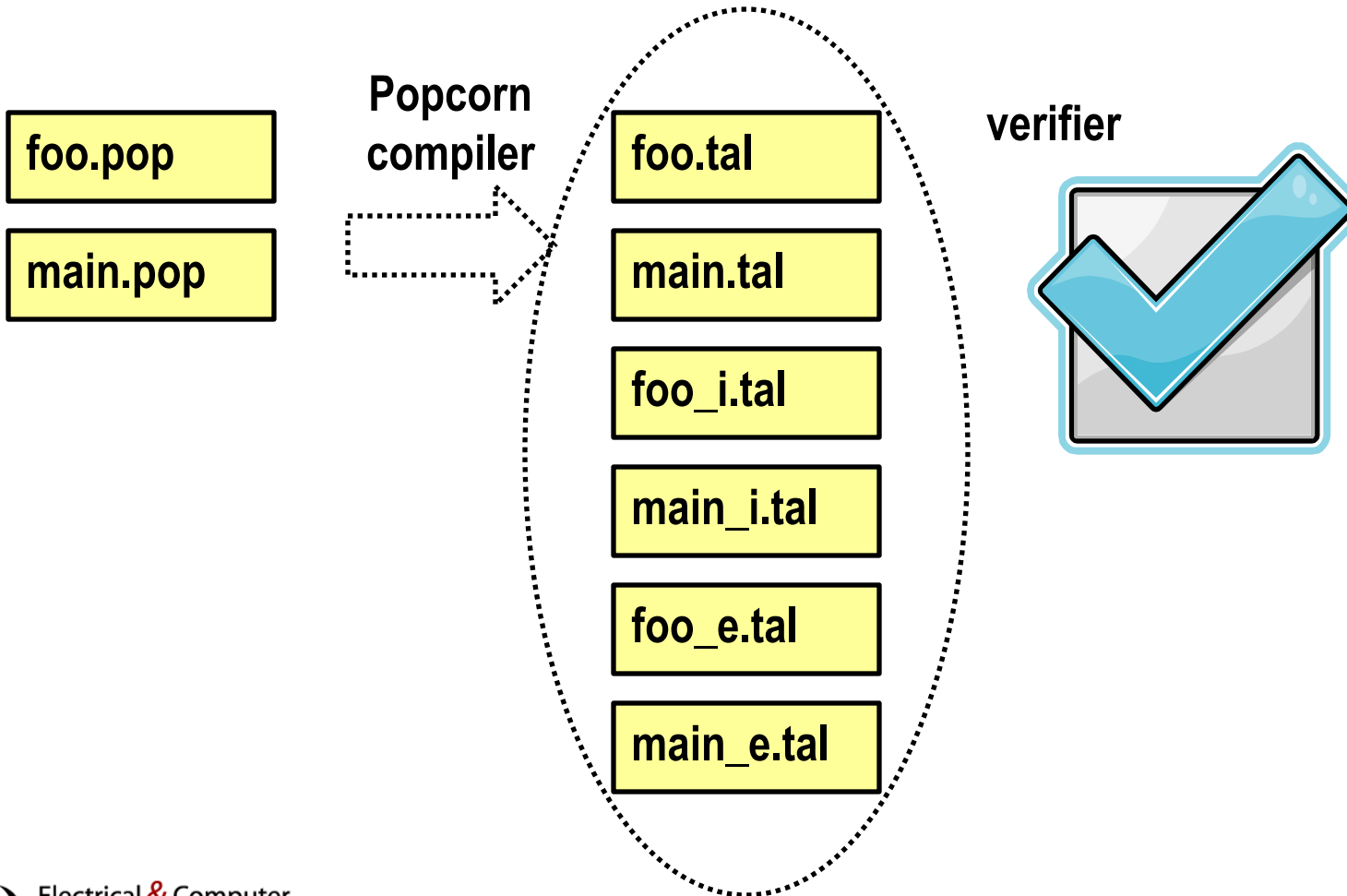
Practice:

- most of IA32 (32-bit Intel x86)
- more type constructors
 - everything you can think of and more
- safe C compiler
 - ~40,000 LOC & compiles itself

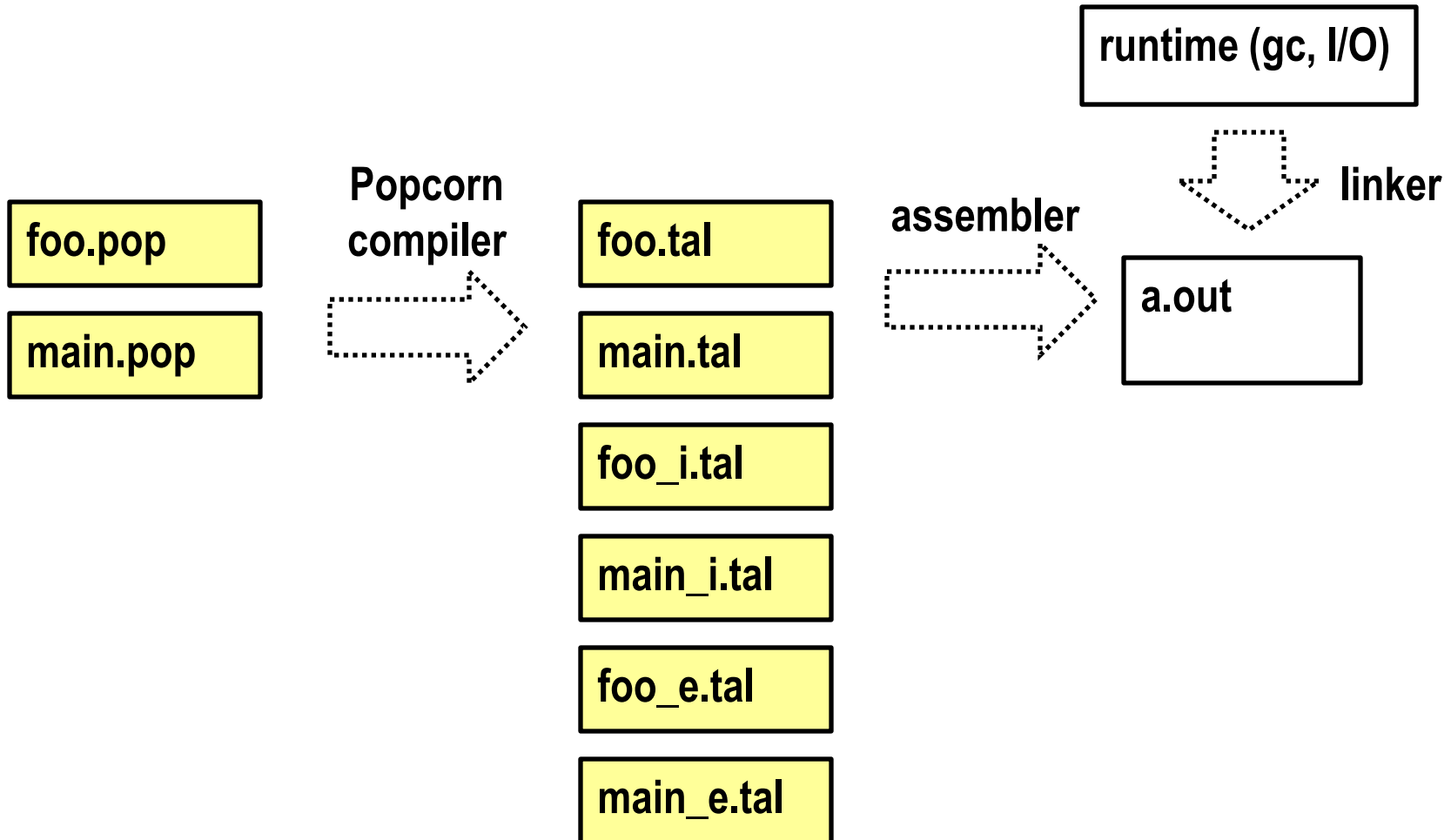
TAL Certifying Compilation



TAL Tools, Compilation Procedure



TAL Tools, Compilation Procedure



Source Language: Tiny

- Types

$$\tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2$$

- Expressions

$$e ::= x \mid f \mid e_1 + e_2 \mid e_1 e_2 \mid \dots$$

- Function declarations

$$D ::= \text{fun } f(x:\tau_1) : \tau_2 = e$$

- Programs

$$P ::= \text{letrec } d_1 \dots d_n \text{ in } e$$

Certifying Compilation

- Series of type-preserving transformations

- Term translation

$$\begin{aligned} \mathcal{E} [[e_1 + e_2]] = & \mathcal{E} [[e_1]] \\ & \text{push } r_a \\ & \mathcal{E} [[e_1]] \\ & \text{pop } r_t \\ & \text{add } r_a, r_t, r_a \end{aligned}$$

- Type translation

- Desired property:

If P is a well-typed Tiny program then the compiled program $\mathcal{P} [[P]]$ is also well-typed

Rest of the Lecture: Examples

- TAL core types:
 - bytes, tuples, code
- Control flow:
 - calling conventions, stacks, exns
- I won't get to:
 - closures, objects, modules, type analysis, ADTs

Simple Built-In Types

- Bytes: b1, b2, b4
- Tuples: (τ_1, \dots, τ_n)
- Code: $\{r_1:\tau_1, \dots, r_n:\tau_n\}$
 - like a pre-condition
 - argument type of function
 - no return type because code doesn't really return, just jumps somewhere else...
- Polymorphic types: $\forall \alpha. \tau$

Simple Loop

sum: {ecx:b4, ebx:{eax:b4}}	; int sum(int x) {
mov eax,0	; int a = 0;
jmp test	;
loop: {eax:b4, ecx:b4, ebx:{eax:b4}}	; while(!x) {
add eax,ecx	; a += x;
dec ecx	; x--;
FALLTHRU	; }
test: {eax:b4, ecx:b4, ebx:{eax:b4}}	;
cmp ecx,0	;
jne loop	; return(a);
jmp ebx	; }

Allocation

mkpair: {eax:b4, ebx:{eax:(b4, b4)}}

```
mov      ecx,eax
MALLOC  eax,8,(b4, b4) ; eax : (b4, b4)
mov      [eax+0],ecx ; eax : (b4, b4)
mov      [eax+4],ecx ; eax : (b4, b4)
jmp      ebx
```

Callee-Saves Register

addone: $\forall \alpha. \{ \text{eax:b4}, \text{ecx}:\alpha, \text{ebx}:\{ \text{eax:b4}, \text{ecx}:\alpha \} \}$

```
inc    eax    ; x+1
jmp    ebx    ; return
```

main: $\{ \text{ebx}:\{ \text{eax:b4} \} \}$

```
mov    eax,3
mov    ecx,ebx ; save main's return address
mov    ebx,done
jmp    addone[ $\{ \text{eax:b4} \}$ ]
```

done: $\{ \text{eax:b4}, \text{ecx}:\{ \text{eax:b4} \} \}$

```
inc    eax
jmp    ecx
```

In General...

Need to save more stuff (e.g., locals):

```

MALLOC      ecx,4n,( $\tau_1,\dots,\tau_n$ ) ; frame for storage
mov         [ecx+0],r1
           ... ; save locals
mov         [ecx+4n-4],rn
jmp         addone[( $\tau_1,\dots,\tau_n$ )]

```

Stacks

Want to model the stack

Stack types:

$$\sigma ::= \text{nil} \mid \tau :: \sigma \mid \rho$$

Typing Stack Operations

 $\{ \text{esp} : \sigma \}$
`sub esp, i*4`
 $\{ \text{esp} : b4 :: b4 :: \dots :: b4 :: \sigma \}$
 $\{ \text{esp} : \tau_1 :: \tau_2 :: \dots :: \tau_i :: \sigma \}$
`add esp, i*4`
 $\{ \text{esp} : \sigma \}$
 $\{ r : \tau, \text{esp} : \tau_1 :: \tau_2 :: \dots :: \tau_i :: \sigma \}$
`mov [esp+i*4], r`
 $\{ r : \tau, \text{esp} : \tau_1 :: \tau_2 :: \dots :: \tau :: \sigma \}$
 $\{ r : \tau, \text{esp} : \sigma \}$
`push r`
 $\{ r : \tau, \text{esp} : \tau :: \sigma \}$
 $\{ \text{esp} : \tau_1 :: \tau_2 :: \dots :: \tau_i :: \sigma \}$
`mov r, [esp+i*4]`
 $\{ r : \tau_i, \text{esp} : \tau_1 :: \tau_2 :: \dots :: \tau_i :: \sigma \}$
 $\{ \text{esp} : \tau :: \sigma \}$
`pop r`
 $\{ r : \tau, \text{esp} : \sigma \}$

Recursion Through Stack Variables

fact: $\forall \rho. \{ \text{eax:b4}, \text{esp}:\{\text{eax:b4}, \text{esp}:\rho\}::\rho \}$

 cmp eax,1

 jne L[ρ]

 retn

L: $\forall \rho'. \{ \text{eax:b4}, \text{esp}:\{\text{eax:b4}, \text{esp}:\rho'\}::\rho' \}$

 push eax

 dec eax

 call fact[b4:: $\{ \text{eax:b4}, \text{esp}:\rho' \}::\rho'$]

 pop ecx

 imul eax,ecx

 retn

Fact Fact

fact: $\forall \rho. \{ \text{eax:b4}, \text{esp}:\{\text{eax:b4}, \text{esp}:\rho\}::\rho \}$

Because ρ is abstract, fact cannot read or write this portion of the stack.

Caller's frame is protected from callee...

Other TAL Features

- Module system
 - interfaces, implementations, ADTs
- Sum type/datatype support
- Fancy arrays/vector typing
- (Higher Order) Type constructors
- Fault tolerance checking
- Other people still writing papers about more
...

Other people still writing papers ...

- Fault-tolerant typed assembly language
(Perry et al., 2007)
- A typed assembly language for confidentiality
(Yu and Islam, 2006)
- Toward a foundational typed assembly language
(Crary, 2003)
- Non-interference for a typed assembly language
(Medel et al., 2005)

Long-term Plans

Low-level, portable, safe language:

- OO-support of Java
- typing support of ML
- programmer control of C
 - good model of space
 - good model of running time
 - many optimizations expressible in the language

Microsoft research working on a new compiler
(Phoenix) to generate TAL

Summary

- Types provide a syntactic framework for enforcing abstraction
 - static typing holds the promise of cheap security enforcement
- But until recently, had to buy into high-level languages to get static typing
 - performance issues
 - TCB issues
- TAL concentrates on orthogonal typing constructs that can be used to encode high-level language or compiler invariants

Sources

- Slides from Dave Walker's COS 598E: Foundations of Language-Based Security at Princeton
- Morrissett et al. TALx86: A realistic typed assembly language. In the *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, 1999.