Typed Assembly Language (TAL)

Lujo Bauer

18-732

Spring 2015
Recently: Types and type-safety

- Types specify expected program behavior
- Semantics define what safe behavior is
- Type-safety:
  Typing => always follow semantics => always safe behavior

- Type-checking is at the level of source code
  - Code consumer must trust code producer
  - Compiler must be in the TCB (what if it has a bug?)
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

(where “abstraction” ≈ “type”)
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

Source code → Certifying compiler → Typed assembly code

Code producer

Typed assembly code → Verifier → Assembler + linker → Machine code

Code consumer

= must be trusted
Traditional Compilation

Source code → Verifier → Compiler → Assembler + linker → Machine code

Code producer

Code consumer

Machine code

= must be trusted
TAL Tools, Compilation Procedure

```
foo.pop  main.pop

foo.tal  main.tal
foo_i.tal main_i.tal
foo_e.tal main_e.tal
```

Popcorn compiler

verifier
TAL Tools, Compilation Procedure

- foo.pop
- main.pop

Popcorn compiler

- foo.tal
- main.tal
- foo_i.tal
- main_i.tal
- foo_e.tal
- main_e.tal

assembler

linker

runtime (gc, I/O)

a.out
Source Language: Tiny

- **Types**
  \[ \tau ::= \text{int} | \tau_1 \rightarrow \tau_2 \]

- **Expressions**
  \[ e ::= x | f | e_1 + e_2 | e_1 \cdot e_2 | \ldots \]

- **Function declarations**
  \[ D ::= \text{fun} \; f(x:\tau_1) : \tau_2 = e \]

- **Programs**
  \[ P ::= \text{letrec} \; d_1 \ldots d_n \; \text{in} \; e \]
Certifying Compilation

- **Series of type-preserving transformations**
  - Term translation
    \[
    \varepsilon[[e_1+e_2]] = \varepsilon[[e_1]] \\
    \text{push } r_a \\
    \varepsilon[[e_2]] \\
    \text{pop } r_t \\
    \text{add } r_a, r_t, r_a
    \]
  - Type translation

- **Desired property:**
  If \( P \) is a well-typed Tiny program then the compiled program \( P[[P]] \) is also well-typed
Rest of the Lecture: Examples of Assembly-level Types

- **TAL core types:**
  - bytes, tuples, code

- **Control flow:**
  - calling conventions, stacks, exceptions

- **Won’t get to:**
  - closures, objects, modules, type analysis, ADTs
Simple Built-In Types $\tau$

- **Bytes:** $b1$, $b2$, $b4$
- **Tuples:** $(\tau_1, \cdots, \tau_n)$
- **Code:** $\{r_1: \tau_1, \cdots, r_n: \tau_n\}$
  - 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

```
; int sum(int x) {
;    int a = 0;
;    while(x!=0) {
;        a += x;
;        x--;
;    }
;    return(a);
; }
```
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: $\forall \alpha. \{eax:b4, ecx:\alpha, ebx:\{eax:b4, ecx:\alpha\}\}$
  inc eax ; x+1
  jmp ebx ; return

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

done: $\{eax:b4, ecx:\{eax:b4\}\}$
  inc eax
  jmp ecx
In General…

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

MALLOC ecx,4n, (τ₁,…,τₙ) ; frame for storage
mov [ecx+0],r1

... ; save locals
mov [ecx+4n-4],rn
jmp addone[((τ₁,…,τₙ))]
Stacks $\sigma$

Want to model the stack

Stack types:

$\sigma ::= nil \mid \tau::\sigma \mid \rho$
Typing Stack Operations

{ esp: ρ }
sub esp,i*4
{ esp: b4::b4:: ... ::b4::ρ }

{ r: τ, esp: τ1::τ2:: ... ::τi::ρ }
mov [esp+i*4],r
{ r: τ, esp: τ1::τ2:: ... ::τ::ρ }

{ esp: τ1::τ2:: ... ::τi::ρ }
mov r,[esp+i*4]
{ r: τi, esp: τ1::τ2:: ... ::τi::ρ }

{ esp: τ1::τ2:: ... ::τi::ρ }
add esp,i*4
{ esp : ρ }

{ r: τ, esp: ρ }
push r
{ r: τ, esp: τ::ρ }

{ esp: τ::ρ }
pop r
{ r: τ, esp: ρ }


Recursion Through Stack Variables

First element on stack is pointer to code with type `{…}`

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

  cmp    eax,1
  jne    L[\rho]
  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

\[ \forall \rho. \{ \text{eax}: b4, \ \text{esp}: \{ \text{eax}: b4, \ \text{esp}: \rho \} :: \rho \} \]

Because \( \rho \) 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 Phoenix compiler 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