Typed Assembly Language

Evari Koppel and Magnus Leppik

About presentation

- Why type checking?
- Proof-carrying code
- Typed Assembly Language
- TAL-0
- TAL-1
- Examples

What is TAL

- TAL is assembly language that is extended to make use of annotating datatypes for each used value
- Type checker – to check how code acts when executed
- Type safety – prevent type errors

Why Typing Checking

- Low-level code and high-level program
- Type checking – convenient way to ensure that a program has certain semantic properties
- Major component of the security infrastructure in distributed systems
- Memory safety
- Type safety
- Malicous code

Proof-carrying code (1)

- The principle of PCC is that the need to trust a piece of code is eliminated by machine-checkable proof that the code has certain properties.
- Using PCC to build trustworthy systems:
  - What properties should we require of the code?
  - How do code producers construct a formal proof that their code has the desired properties?

Proof-carrying code (2)

- Solution: type-preserving compilation.
- We seek a principled approach to the design of typed intermediate language.

TAL-0: Control-Flow-Safety(1)

- Control-Flow safety
- Focus on control-flow safety will let us start with simple abstract machine
- The syntax for control-flow-safety assembly language:

r	::=	*registers:*	ι	::=	*instructions:*
	`r1 | r2 | ··· | rk`			r_d`:=`v	
v	::=	*operands:*		r_d`:=`r_s`+`v	
	n	*integer literal*		`if` r `jump` v	
	ℓ	*label or pointer*	I	::=	*instruction sequences:*
	r	*registers*		`jump` v	
				ι`;`I	

TAL-0: Control-Flow-Safety(2)

- We model evaluation of TAL-0 assembly programs using a rewriting relation between *abstract* machine states.
- We maintain the distinction between labels and arbitrary integers.
- Enforcing the safety property now reduces to ensuring that abstract machine cannot get stuck.

TAL-0: Control-Flow-Safety(3)

- Syntax for TAL-0 abstract machines:

$\mathcal{R}$ ::=		*register files:*	H ::=		*heaps:*
$\{r_1 = v_1, \ldots, r_k = v_k\}$			$\{\ell_1 = h_1, \ldots, \ell_m = h_m\}$		
h ::=		*heap values:*	M ::=		*machine states:*
I		*code*	(H, R, I)		

TAL-0: Control-Flow-Safety(4)

- Rewriting rules for TAL-0:

$$\frac{H(\hat{R}(v)) = I}{(H, R, \texttt{jump } v) \longrightarrow (H, R, I)} \quad \text{(JUMP)}$$

$$(H, R, r_d := v; I) \longrightarrow (H, R[r_d = \hat{R}(v)], I) \quad \text{(MOV)}$$

$$\frac{R(r_s) = n_1 \qquad \hat{R}(v) = n_2}{(H, R, r_d := r_s + v; I) \longrightarrow (H, R[r_d = n_1 + n_2], I)} \quad \text{(ADD)}$$

$$\frac{R(r) = 0 \qquad H(\hat{R}(v)) = I'}{(H, R, \texttt{if } r \texttt{ jump } v; I) \longrightarrow (H, R, I')} \quad \text{(IF-EQ)}$$

$$\frac{R(r) = n \qquad n \neq 0}{(H, R, \texttt{if } r \texttt{ jump } v; I) \longrightarrow (H, R, I)} \quad \text{(IF-NEQ)}$$

TAL-0 Type System (1)

- Goal: ensure that any well-formed abstract machine M cannot get stuck.
- Our type system has to:
  - Distinguish labels from integers
    - Ensures that operands of a control transfer are labels
  - No matter how many steps are taken by M, it never gets into a stuck state (i.e typing preserved)

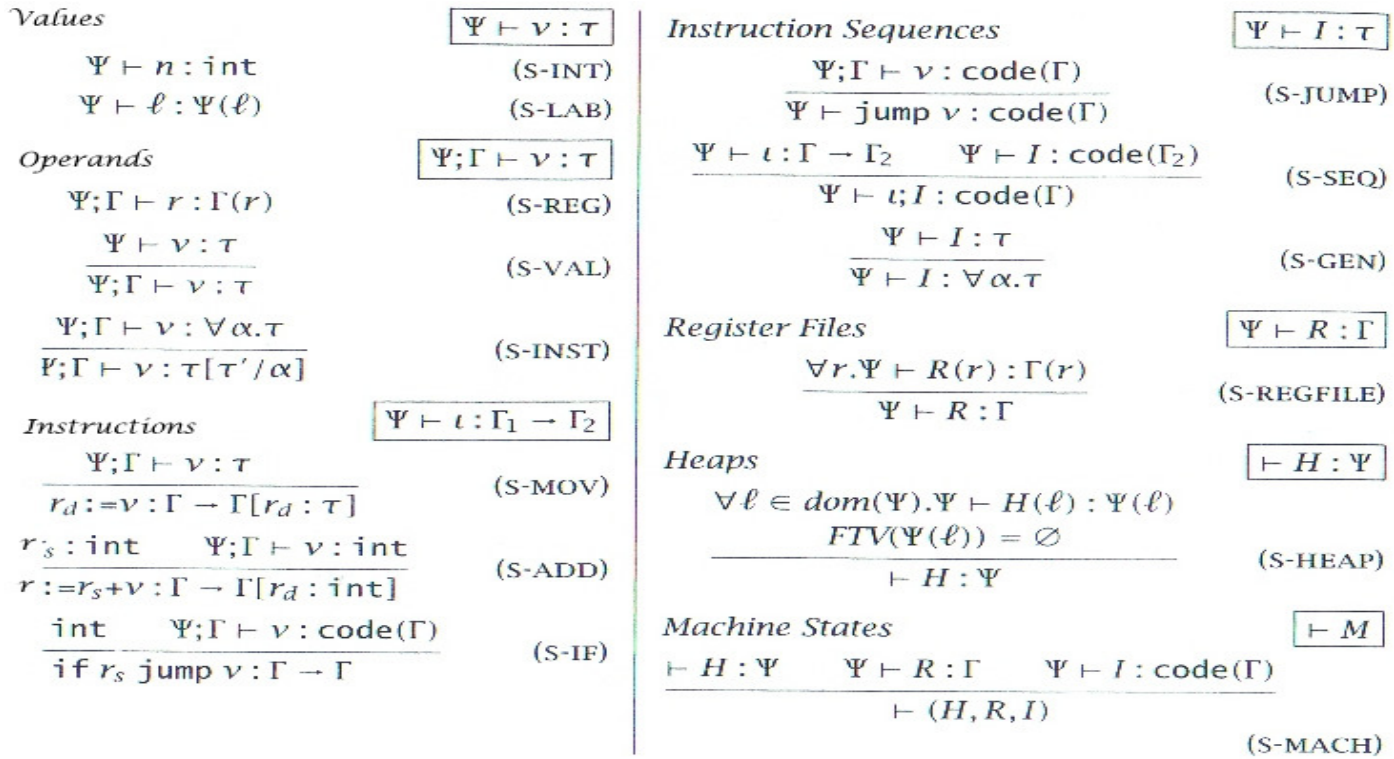
TAL-0 Type System (2)

- Type syntax:

τ	::=	*operand types:*	Γ	::=	*register file types:*
	$\texttt{int}$	*word-sized integers*		$\{r_1 : \tau_1, \ldots, r_k : \tau_k\}$	
	$\texttt{code}(\Gamma)$	*code labels*	Ψ	::=	*heap types:*
	α	*type variables*		$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$	
	$\forall \alpha.\tau$	*universal polymorphic types*			

TAL-0 Type System (3)

- We now formalize the type system using the inference rules:



Values $\boxed{\Psi \vdash v : \tau}$

$$\Psi \vdash n : \mathtt{int} \quad \text{(S-INT)}$$

$$\Psi \vdash \ell : \Psi(\ell) \quad \text{(S-LAB)}$$

Operands $\boxed{\Psi;\Gamma \vdash v : \tau}$

$$\Psi;\Gamma \vdash r : \Gamma(r) \quad \text{(S-REG)}$$

$$\frac{\Psi \vdash v : \tau}{\Psi;\Gamma \vdash v : \tau} \quad \text{(S-VAL)}$$

$$\frac{\Psi;\Gamma \vdash v : \forall\alpha.\tau}{\Psi;\Gamma \vdash v : \tau[\tau'/\alpha]} \quad \text{(S-INST)}$$

Instructions $\boxed{\Psi \vdash \iota : \Gamma_1 \to \Gamma_2}$

$$\frac{\Psi;\Gamma \vdash v : \tau}{r_d := v : \Gamma \to \Gamma[r_d : \tau]} \quad \text{(S-MOV)}$$

$$\frac{r_s : \mathtt{int} \qquad \Psi;\Gamma \vdash v : \mathtt{int}}{r := r_s + v : \Gamma \to \Gamma[r_d : \mathtt{int}]} \quad \text{(S-ADD)}$$

$$\frac{\mathtt{int} \qquad \Psi;\Gamma \vdash v : \mathtt{code}(\Gamma)}{\mathtt{if}\ r_s\ \mathtt{jump}\ v : \Gamma \to \Gamma} \quad \text{(S-IF)}$$

Instruction Sequences $\boxed{\Psi \vdash I : \tau}$

$$\frac{\Psi;\Gamma \vdash v : \mathtt{code}(\Gamma)}{\Psi \vdash \mathtt{jump}\ v : \mathtt{code}(\Gamma)} \quad \text{(S-JUMP)}$$

$$\frac{\Psi \vdash \iota : \Gamma \to \Gamma_2 \qquad \Psi \vdash I : \mathtt{code}(\Gamma_2)}{\Psi \vdash \iota; I : \mathtt{code}(\Gamma)} \quad \text{(S-SEQ)}$$

$$\frac{\Psi \vdash I : \tau}{\Psi \vdash I : \forall\alpha.\tau} \quad \text{(S-GEN)}$$

Register Files $\boxed{\Psi \vdash R : \Gamma}$

$$\frac{\forall r.\Psi \vdash R(r) : \Gamma(r)}{\Psi \vdash R : \Gamma} \quad \text{(S-REGFILE)}$$

Heaps $\boxed{\vdash H : \Psi}$

$$\frac{\forall \ell \in dom(\Psi).\Psi \vdash H(\ell) : \Psi(\ell) \qquad FTV(\Psi(\ell)) = \varnothing}{\vdash H : \Psi} \quad \text{(S-HEAP)}$$

Machine States $\boxed{\vdash M}$

$$\frac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \Psi \vdash I : \mathtt{code}(\Gamma)}{\vdash (H, R, I)} \quad \text{(S-MACH)}$$

Proof of Type Soundness for TAL-0 (1)

- It suffices to show:
  - Well-typed machine state is not immediately stuck (progress)
  - When it steps to a new machine state M', that state is also well-typed (preservation).

Proof of Type Soundness for TAL-0 (2)

LEMMA [TYPE SUBSTITUTION]: If:

1. $\Psi;\Gamma \vdash v : \tau_1$, then $\Psi;\Gamma[\tau/\alpha] \vdash v : \tau_1[\tau/\alpha]$.
2. $\Psi \vdash \iota : \Gamma_1 \to \Gamma_2$ then $\Psi \vdash \iota : \Gamma_1[\tau/\alpha] \to \Gamma_2[\tau/\alpha]$.
3. $\Psi \vdash I : \tau_1$, then $\Psi \vdash I : \tau_1[\tau/\alpha]$.
4. $\Psi \vdash R : \Gamma$, then $\Psi \vdash R : \Gamma[\tau/\alpha]$.

Proof of Type Soundness for TAL-0 (3)

LEMMA [REGISTER SUBSTITUTION]: If $\vdash H : \Psi$, $\Psi \vdash R : \Gamma$ and $\Psi;\Gamma \vdash v : \tau$ then $\Psi;\Gamma \vdash \hat{R}(v) : \tau$ □

LEMMA [CANONICAL VALUES]: If $\vdash H : \Psi$ and $\Psi \vdash v : \tau$ then:

1. If $\tau = \texttt{int}$ then $v = n$ for some n.
2. If $\tau = \texttt{code}(\Gamma)$ then $v = \ell$ for some $\ell \in dom(H)$ and $\Psi \vdash H(\ell) : \texttt{code}(\Gamma)$. □

Proof of Type Soundness for TAL-0 (4)

LEMMA [CANONICAL OPERANDS]: If $\vdash H : \Psi$, $\Psi \vdash R : \Gamma$, and $\Psi;\Gamma \vdash v : \tau$ then:

1. If $\tau = \texttt{int}$ then $\hat{R}(v) = n$ for some n.
2. If $\tau = \texttt{code}(\Gamma)$ then $\hat{R}(v) = \ell$ for some $\ell \in dom(H)$ and $\Psi \vdash H(\ell) : \texttt{code}(\Gamma)$.

□

THEOREM [SOUNDNESS OF TAL-0]: If $\vdash M$, then there exists an M' such that $M \longrightarrow M'$ and $\vdash M'$.

□

Proof Representation and Checking

- For TAL-0 it is sufficient to provide types for the labels;
- Keep the type checker as simple as possible:

a) Type reconstruction is entirly syntax directed (for any given term at most one rule should apply)

b) Explicit representation of the complete proof of well-formedness

  - We can ship the binary machine code, disassemble it and then compare it against the assembly-level proof (proof-carrying code)

TAL-1: Simple Memory-Safety (1)

- TAL-0 includes registers and heap-allocated code; no support for allocated *data*.
- TAL-1:
  - adds primitive support for allocated objects that can be shared by reference (i.e pointer)
  - includes a notion of object-level memory safety.

- How to accomodate locations that hold values of different types at different times?

TAL-1: Simple Memory-Safety (2)

```
     {r1:ptr(code(...))}
1.   r3 := 0;
2.   Mem[r1] := r3;
3.   r4 := Mem[r1];
4.   jump r4
```

- The code above should be rejected by the type-checker (control-flow safety property)

```
     {r1:ptr(code(...)),r2:ptr(code(...))}
1.   r3 := 0;
2.   Mem[r1] := r3;
3.   r4 := Mem[r2];
4.   jump r4
```

TAL-1: Simple Memory-Safety (3)

- We need some support for
  - Allocating and initializing data structures that are to be shared;
  - Stack-allocating procedure frames.

- Separate locations into two classes:
  - Shared pointers that support arbitrary aliasing;
  - Unique pointers that will support updates that change the type of the contents.

The TAL-1 Extended Abstract Machine (1)

Syntactic extensions to TAL-0 and rewriting rules:

$r ::=$	*registers:*
r_1 | r2 | $\cdots$ | rk	*gp registers*
	stack pointer
$i ::=$	*instructions:*
...	*as in TAL-0*
$r := \texttt{Mem}[r_s + n]$	*load from memory*
$\texttt{Mem}[r_d + n] := r_s$	*store to memory*
$r := \texttt{malloc}\ n$	*allocate n heap words*
$\texttt{Commit}\ r_d$	*become shared*
$\texttt{salloc}\ n$	*allocate n stack words*

The TAL-1 Extended Abstract Machine (2)

		`sfree` n	*free n stack words*
v	::=		*operands:*
		r	*registers*
		n	*integer literals*
		ℓ	*code or shared data pointers*
		`uptr`(h)	*unique data pointers*
h	::=		*heap values:*
		I	*instruction sequences*
		$\langle v_1, \ldots, v_n \rangle$	*tuples*

The TAL-1 Extended Abstract Machine (3)

- The rewriting rules for the instructions of TAL-1

$$\frac{\hat{R}(v) \neq \mathtt{uptr}(h)}{(H, R, r_d := v; I) \longrightarrow (H, R[r_d = v], I)} \quad \text{(MOV-1)}$$

This rule can only fire when the source operand is not a unique pointer.

We must now give the rewriting rules for the new instructions:

$$(H, R, r_d := \mathtt{malloc}\ n; I) \longrightarrow (H, R[r_d = \mathtt{uptr}\langle m_1, \ldots, m_n \rangle], I) \quad \text{(MALLOC)}$$

$$\frac{r_d \neq \mathtt{sp} \qquad \ell \notin dom(H)}{(H, R[r_d = \mathtt{uptr}(h)], \mathtt{commit}\ r_d; I) \longrightarrow (H[\ell = h], R[r_d = \ell], I)} \quad \text{(COMMIT)}$$

The TAL-1 Extended Abstract Machine (4)

$$\frac{R(r_s) = \ell \qquad H(\ell) = \langle v_0, \ldots, v_n, \ldots, v_{n+m} \rangle}{(H, R, r_d := \mathsf{Mem}[r_s + n]; I) \longrightarrow (H, R[r_d = v_n], I)} \quad \text{(LD-S)}$$

$$\frac{R(r_s) = \mathsf{uptr}\langle v_0, \ldots, v_n, \ldots, v_{n+m} \rangle}{(H, R, r_d := \mathsf{Mem}[r_s + n]; I) \longrightarrow (H, R[r_d = v_n], I)} \quad \text{(LD-U)}$$

$$\frac{R(r_d) = \ell \qquad H(\ell) = \langle v_0, \ldots, v_n, \ldots, v_{n+m} \rangle \qquad R(r_s) = v \qquad v \neq \mathsf{uptr}(h)}{(H, R, \mathsf{Mem}[r_d + n] := r_s; I) \longrightarrow (H[\ell = \langle v_0, \ldots, v, \ldots, v_{n+m} \rangle], R, I)} \quad \text{(ST-S)}$$

The TAL-1 Extended Abstract Machine (5)

$$\frac{R(r_d) = \mathsf{uptr}\langle v_0, \ldots, v_n, \ldots, v_{n+m}\rangle, \quad R(r_s) = v \quad v \neq \mathsf{uptr}(n)}{(H, R, \mathsf{Mem}[r_d + n] := r_s; I) \rightarrow (H, R[r_d = \mathsf{uptr}\langle v_0, \ldots, v, \ldots, v_{n+m}\rangle], I)} \quad \text{(ST-U)}$$

$$\frac{R(\mathsf{sp}) = \mathsf{uptr}\langle v_0, \ldots, v_p\rangle \quad p + n \leq \mathsf{MAXSTACK}}{(H, R, \mathsf{salloc}\ n) \rightarrow (H, R[\mathsf{sp} = \mathsf{uptr}\langle m_1, \ldots, m_n, v_0, \ldots, v_p\rangle])} \quad \text{(SALLOC)}$$

$$\frac{R(\mathsf{sp}) = \mathsf{uptr}\langle m_1, \ldots, m_n, v_0, \ldots, v_p\rangle}{(H, R, \mathsf{sfree}\ n) \rightarrow (H, R[\mathsf{sp} = \mathsf{uptr}\langle v_0, \ldots, v_p\rangle])} \quad \text{(SFREE)}$$

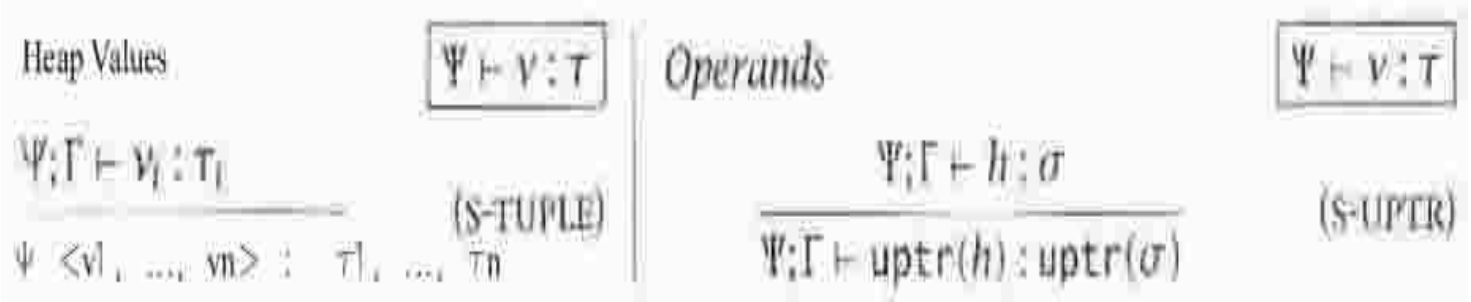
TAL-1 Changes to the Type System (1)

- New set of types for classifying TAL-1 values and new typing rules:

τ ::=	*operand types:*	σ ::=	*allocated types:*
...	*as in TAL-0*	ϵ	*empty*
$\mathtt{ptr}(\sigma)$	*shared data pointers*	τ	*value type*
$\mathtt{uptr}(\sigma)$	*unique data pointers*	σ_1, σ_2	*adjacent*
$\forall \rho.\tau$	*quantification over allocated types*	ρ	*allocated type variable*

TAL-1 Changes to the Type System (2)

- New typing rules:



TAL-1 Changes to the Type System (3)

- New typing rules:

Instructions

$$\boxed{\Psi \vdash \iota : \Gamma_1 \to \Gamma_2}$$

$$\frac{\Psi;\Gamma \vdash v : \tau \qquad \tau = \texttt{uptr}(\sigma)}{\Psi \vdash r_d := v : \Gamma \to \Gamma[r_d : \tau]} \quad \text{(S-MOV-1)}$$

$$\frac{n \geq 0}{\Psi \vdash r_d := \texttt{malloc}\ n : \Gamma \to \Gamma[r_d : \texttt{uptr}(\underbrace{\texttt{int}, \ldots, \texttt{int}}_{n})]} \quad \text{(S-MALLOC)}$$

$$\frac{\Psi;\Gamma \vdash r_d : \texttt{uptr}(\sigma) \qquad r_d \neq \texttt{sp}}{\Psi \vdash \texttt{commit}\ r_d : \Gamma \to \Gamma[r_d : \texttt{ptr}(\sigma)]} \quad \text{(S-COMMIT)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \texttt{ptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash r_d := \texttt{Mem}[r_s + n] : \Gamma \to \Gamma[r_d : \tau_n]} \quad \text{(S-LDS)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \texttt{uptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash r_d := \texttt{Mem}[r_s + n] : \Gamma \to \Gamma[r_d : \tau_n]} \quad \text{(S-LDU)}$$

TAL-1 Changes to the Type System (4)

$$\frac{\Psi;\Gamma \vdash r_s : \tau_n \qquad \tau_n \neq \mathtt{uptr}(\sigma') \qquad \Psi;\Gamma \vdash r_d : \mathtt{ptr}(\tau_1,\ldots,\tau_n,\sigma)}{\Psi \vdash \mathtt{Mem}[r_d + n] := r_s : \Gamma \to \Gamma} \quad \text{(S-STS)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \tau \qquad \tau \neq \mathtt{uptr}(\sigma') \qquad \Psi;\Gamma \vdash r_d : \mathtt{uptr}(\tau_1,\ldots,\tau_n,\sigma)}{\Psi \vdash \mathtt{Mem}[r_d + n] := r_s : \Gamma \to \Gamma[r_d : \mathtt{uptr}(\tau_1,\ldots,\tau,\sigma)]} \quad \text{(S-STU)}$$

$$\frac{\Psi;\Gamma \vdash \mathtt{sp} : \mathtt{uptr}(\sigma) \qquad n \geq 0}{\Psi \vdash \mathtt{salloc}\, n : \Gamma \to \Gamma[\mathtt{sp} : \mathtt{uptr}(\underbrace{\mathtt{int},\ldots,\mathtt{int}}_{n},\sigma)]} \quad \text{(S-SALLOC)}$$

$$\frac{\Psi;\Gamma \vdash \mathtt{sp} : \mathtt{uptr}(\tau_1,\ldots,\tau_n,\sigma)}{\Psi \vdash \mathtt{sfree}\, n : \Gamma \to \Gamma[\mathtt{sp} : \mathtt{uptr}(\sigma)]} \quad \text{(S-SFREE)}$$

- At this point TAL-1 provides enough mechanism for the compiler of a polymorphic, procedural language.

Compiling to TAL-1 (1)

- A simple example:

```
int prod (int x, int y){
  int a = 0;
  while (x != 0) {
    a = a + y;
    x = x – 1;
  }
  return a;
}
```

Compiling to TAL-1 (2)

```
prod: ∀a,b,c,s.
       code{r1:a,r2:b,r3:c,sp:uptr(int,int,s),
            r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      r2 := Mem[sp];     // r2:int, r2 := x
      r3 := Mem[sp+1];   // r3:int, r3 := y
      r1 := 0            // r1:int, a := 0
      jump loop

loop: ∀s.code{r1,r2,r3:int,sp:uptr(int,int,s),
              r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      if r2 jump done;   // if x ↔ 0 goto done
      r1 := r1 + r3;     // a := a + y
      r2 := r2 + (-1);   // x := x - 1
      jump loop

done: ∀s.code{r1,r2,r3:int,sp:uptr(int,int,s),
              r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      sfree 2;           // sp:uptr(s)
      jump r4
```

Compiling to TAL-1 (3)

```
prod: ∀a,b,c,s.
       code{r1:a,r2:b,r3:c,sp:uptr(int,int,s),
            r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      r2 := Mem[sp];     // r2:int, r2 := x
      r3 := Mem[sp+1];   // r3:int, r3 := y
      r1 := 0            // r1:int, a := 0
      jump loop
```

Compiling to TAL-1 (4)

```
loop: ∀s.code{r1,r2,r3:int,sp:uptr(int,int,s),
              r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      if r2 jump done;  // if x ↔ 0 goto done
      r1 := r1 + r3;    // a := a + y
      r2 := r2 + (-1);  // x := x - 1
      jump loop
```

Compiling to TAL-1 (5)

```
done: ∀s.code{r1,r2,r3:int,sp:uptr(int,int,s),
              r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      sfree 2;          // sp:uptr(s)
      jump r4
```

Some Real World Issues

- TAL-1 and the extensions described earlier provide mechanisms needed to implement only very simple languages.
- Further extensions to TAL-1:
  - STAL;
  - TALT;
  - TALx86.

Conclusion

- The typing annotations are produced and consumed by machines;
- Low-level languages present new challanges to type system designers;
- Ideally, proofs should be carried out in a machine-checked environment.

Used materials

Benjamin C. Pierce edition

"Advanced Topics in Types and Programming Languages “ (MIT Press, 2005)

- Typed Assembly Language, by Greg Morrisett

Thank you!