A Type System for TAL-0

Software Security

Spring 2025

The material in these notes has been adapted from Section 4 of [1].

Types

The goal of the type system presented here is to ensure that when well-typed programs never "get stuck" when run using the abstract machine from last week.

Operand types τ are given by the following BNF grammar:

$$
\begin{array}{rr}
\tau ::= \mathsf{int} & \text{(integers)} \\
\mid \mathsf{code}(\Gamma) & \text{(code labels)} \\
\mid \alpha & \text{(type variables)} \\
\mid \forall \alpha.\tau & \text{(polymorphic types)}
\end{array}
$$

Figure 1: Operand types for TAL-0

We begin by introducing the different sorts of "type" that will be required. First, we require *operand types*, constructed as in Figure 1. Integer values will have type int, and labels will have type $\mathsf{code}(\Gamma)$ where Γ is a *register valuation type*. That is, a total function:

$$\Gamma : \{1, \ldots, k\} \to \mathcal{T}$$

where $\mathcal{T}$ is the set of all operand types τ. The idea here is that label $l \in \mathcal{L}$ has type $\mathsf{code}(\Gamma)$ when the code pointed to by l expects register r_i to store a value of type $\Gamma(i)$ for each $1 \leq i \leq k$. We also support type variables and polymorphic types, which will be discussed a bit later on in these notes[1]. Finally, we require *heap types* ψ, which are partial functions:

$$\psi : \mathcal{L} \to \mathcal{T}$$

[1] It is worth mentioning that in the type $\forall \alpha.\tau$, any occurrences of the variable α in τ are considered *bound*, and that we treat types that are equivalent up to renaming of bound variables as the same type (recall: α-equivalence in the lambda calculus). If this does not make sense to you, do not worry! You should still be able to understand these notes.

The idea here is essentially that a heap H will have type ψ in case for any $l \in \mathcal{L}$ the code $H(l)$ has the type $\psi(l)$.

Typing Rules

While the type system for TAL-0 is, at a glance, quite complicated, most of the concepts being expressed are relatively straightforward. We will go through the different parts of the type system in a "bottom-up" fashion, proceeding from values and operands to abstract machine states. The rules of the type system are summarized in Figure 2.

Values A *value typing judgement* looks like:

$$\psi \vdash v : \tau$$

which we read as the statement that value v has type τ in the context of any heap with type ψ. The rules concerning value typing judgements assert that integer constants have type int and that labels have the type assigned to them by the heap type:

$$\frac{n \in \mathbb{Z}}{\psi \vdash n : \mathsf{int}}\ \text{S-INT} \qquad\qquad \frac{l \in \mathsf{dom}(\psi)}{\psi \vdash l : \psi(l)}\ \text{S-LAB}$$

Operands An *operand typing judgement* looks like:

$$\psi; \Gamma \vdash v : \tau$$

which we read as the statement that operand v has type τ in the context of any heap with heap type ψ and register valuation with register valuation type Γ. Two rules concerning operand typing judgements assert that registers have the type assigned to them by the register valuation type, and that derivable value typing judgements hold in the context of any register valuation type:

$$\frac{i \leq k}{\psi; \Gamma \vdash r_i : \Gamma(i)}\ \text{S-REG} \qquad\qquad \frac{\psi \vdash v : \tau}{\psi; \Gamma \vdash v : \tau}\ \text{S-VAL}$$

There is one more rule concerning operand typing judgements, which allows us to *instantiate* polymorphic types:

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

That is, for v to have type $\forall \alpha.\tau$ in some context means that for any type τ' it is sensible to v as having type $\tau[\tau'/\alpha]$ in that context, where $\tau[\tau'/\alpha]$ is the result of substituting[2] τ' for each instance of the variable α in τ.

[2] In a variable-capture avoiding manner. Again, if this means nothing to you, that's okay!

Instructions An *instruction typing judgement* looks like:

$$\psi \vdash \iota : \Gamma_1 \to \Gamma_2$$

which we read as the statement that in the context of any heap with type ψ and any register valuation with type Γ_1, it is possible to execute the instruction ι, and moreover that this will result in a register valuation of type Γ_2. The rules concerning instruction typing judgements are as follows:

$$\frac{\psi; \Gamma \vdash v : \tau}{\psi \vdash r_i := v : \Gamma \to \Gamma[(i, \tau)]}\ \text{S-MOVE} \qquad \frac{\psi; \Gamma \vdash r_j : \mathsf{int} \qquad \psi; \Gamma \vdash v : \mathsf{int}}{\psi \vdash r_i := r_j + v : \Gamma \to \Gamma[(r_i : \mathsf{int})]}\ \text{S-ADD}$$

$$\frac{\psi; \Gamma \vdash r_i : \mathsf{int} \qquad \psi; \Gamma \vdash v : \mathsf{code}(\Gamma)}{\psi \vdash \mathsf{if}\ r_i\ \mathsf{jump}\ v : \Gamma \to \Gamma}\ \text{S-COND}$$

Of particular interest is the second part of the antecedent of the rule for conditional jumps, which says that in order for a jump instruction to be well-typed in the context of a register valuation with type Γ, the instruction sequence jumped to must expect a register valuation of precisely that type.

Instruction Sequences An *instruction sequence typing judgement* looks like:

$$\psi \vdash I : \tau$$

which, in particular when $\tau = \mathsf{code}(\Gamma)$, we read as the statement that in the context of any heap with type ψ, the instruction sequence I expects a register valuation of type Γ. Two rules concerning instruction sequence typing judgements are as follows:

$$\frac{\psi; \Gamma \vdash v : \mathsf{code}(\Gamma)}{\psi \vdash \mathsf{jump}\ v : \mathsf{code}(\Gamma)}\ \text{S-JUMP} \qquad \frac{\psi \vdash \iota : \Gamma \to \Gamma_2 \qquad \psi \vdash I : \mathsf{code}(\Gamma_2)}{\psi \vdash \iota; I : \mathsf{code}(\Gamma)}\ \text{S-SEQ}$$

There is one more rule concerning instruction sequence typing judgements:

$$\frac{\psi \vdash I : \tau}{\psi \vdash \forall \alpha.\tau}\ \text{S-GEN}$$

Register Valuations A *register valuation typing judgement* looks like:

$$\psi \vdash R : \Gamma$$

which we read as the statement that in the context of any heap with type ψ the register valuation R has type Γ. There is only one rule concerning register valuation typing judgements:

$$\frac{\forall i \in \{1, \ldots, k\}.\psi \vdash R(i) \vdash \Gamma(i)}{\psi \vdash R : \Gamma}\ \text{S-REGVAL}$$

Heaps A *heap typing judgement* looks like:

$$\vdash H : \psi$$

which we read as the statement that heap H has heap type ψ. There is only one rule concerning heap typing judgements:

$$\frac{\forall l \in \mathsf{dom}(\psi).\psi \vdash H(l) : \psi(l) \qquad \forall l \in \mathsf{dom}(\psi).\mathsf{FV}(\psi(l)) = \emptyset}{\vdash H : \psi} \text{ S-HEAP}$$

where the rightmost antecedent insists that each $\psi(l)$ contains no free type varaibles. That is, that each type variable α in $\psi(l)$ is bound by some $\forall\alpha$.

Machine States Finally, a *machine state typing judgement* looks like:

$$\vdash (H, R, I)$$

which we read as the statement that the abstract machine state (H, R, I) is well-typed. There is only one rule concerning machine state typing judgements:

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

So, a machine state (H, R, I) is well-typed in case the heap has some heap type $H : \psi$, in the context of which the register valuation has some register valuation type $R : \Gamma$, such that I is well-typed.

Example

Recall the heap H specifiying our example program from last week:

```
prod: r3 := 0;                          // result = 0
      jump loop

loop: if r1 jump done;                  // if a = 0 we are done
      r3 := r2 + r3;                    // result := result + b
      r1 := r1 + (-1);                  // a := a - 1
      jump loop

done: jump r4                           // return
```

and let $\Gamma : \{1, \ldots, k\} \to \mathcal{T}$ be the following register valuation type:

$$\Gamma = \{(1, \mathsf{int}), (2, \mathsf{int}), (3, \mathsf{int}), (4, \forall\alpha.\mathsf{code}\{(1, \mathsf{int}), (2, \mathsf{int}), (3, \mathsf{int}), (4, \alpha)\})\}$$

Let $\psi : \mathcal{L} \to \mathcal{T}$ be the following heap type:

$\psi = \{(\mathsf{prod}, \mathsf{code}(\Gamma)), (\mathsf{loop}, \mathsf{code}(\Gamma)), (\mathsf{done}, \mathsf{code}(\Gamma)), (\mathsf{exit}, \forall\alpha.\mathsf{code}\{(1, \mathsf{int}), (2, \mathsf{int}), (3, \mathsf{int}), (4, \alpha)\})\}$

We will begin by deriving the following intruction sequence typing judgement:

$$\psi \vdash H(\mathsf{loop}) : \mathsf{code}(\Gamma)$$

That is:

$$\psi \vdash \mathsf{if}\ r_1\ \mathsf{jump\ done}; r_3 := r_2 + r_3; r_1 := r_1 + (-1); \mathsf{jump\ loop} : \mathsf{code}(\Gamma)$$

Observe that it is sufficient to derive all of:

1. $\psi \vdash \mathsf{if}\ r_1\ \mathsf{jump\ done} : \Gamma \to \Gamma$
2. $\psi \vdash r_3 := r_2 + r_3 : \Gamma \to \Gamma$
3. $\psi \vdash r_1 := r_1 + (-1) : \Gamma \to \Gamma$
4. $\psi \vdash \mathsf{jump\ loop} : \mathsf{code}(\Gamma)$

since, we could then obtain $\psi \vdash H(\mathsf{loop}) : \mathsf{code}(\Gamma)$ using the sequencing rule.

We proceed to derive 1-4:

1. We have:

$$\dfrac{\dfrac{1 \leq k}{\psi; \Gamma \vdash \mathsf{int} = \Gamma(1)}\ \text{S-REG} \qquad \dfrac{\dfrac{\mathsf{done} \in \mathsf{dom}(\psi)}{\psi \vdash \mathsf{done} : \mathsf{code}(\Gamma)}\ \text{S-LAB}}{\psi; \Gamma \vdash \mathsf{done} : \mathsf{code}(\Gamma)}\ \text{S-VAL}}{\psi \vdash \mathsf{if}\ r_1\ \mathsf{jump\ done} : \Gamma \to \Gamma}\ \text{S-COND}$$

2. We have:

$$\dfrac{\dfrac{2 \leq k}{\psi; \Gamma \vdash r_2 : \Gamma(2) = \mathsf{int}}\ \text{S-REG} \qquad \dfrac{3 \leq k}{\psi; \Gamma \vdash r_3 : \Gamma(3) = \mathsf{int}}\ \text{S-REG}}{\psi \vdash r_3 := r_2 + r_3 : \Gamma \to \Gamma = \Gamma[(3, \mathsf{int})]}\ \text{S-ADD}$$

3. We have:

$$\dfrac{\dfrac{1 \leq k}{\psi; \Gamma \vdash r_1 : \Gamma(1) = \mathsf{int}}\ \text{S-REG} \qquad \dfrac{\dfrac{(-1) \in \mathbb{Z}}{\psi \vdash (-1) : \mathsf{int}}\ \text{S-INT}}{\psi; \Gamma \vdash (-1) : \mathsf{int}}\ \text{S-VAL}}{\psi \vdash r_1 := r_1 + (-1) : \Gamma \to \Gamma = \Gamma[(1, \mathsf{int})]}\ \text{S-ADD}$$

4. We have:

$$\dfrac{\dfrac{\dfrac{\mathsf{loop} \in \mathsf{dom}(\psi)}{\psi \vdash \mathsf{loop} : \mathsf{code}(\Gamma) = \psi(\mathsf{loop})}\ \text{S-LAB}}{\psi; \Gamma \vdash \mathsf{loop} : \mathsf{code}(\Gamma)}\ \text{S-VAL}}{\psi \vdash \mathsf{jump\ loop} : \mathsf{code}(\Gamma)}\ \text{S-JUMP}$$

And we have therefore derived $\psi \vdash H(\mathsf{loop}) : \mathsf{code}(\Gamma)$.

Concerning Polymorphic Types

A good way to understand the role played by polymorphic types is to consider the following example heap:

$$\begin{array}{ll} \mathsf{foo}: & r_1 := \mathsf{bar}; \\ & \mathsf{jump}\ r_1 \\ \\ \mathsf{bar}: & \ldots \end{array}$$

Without using polymorphic types, any derivation of $\psi \vdash r_1 := \mathsf{bar}; \mathsf{jump}\ r_1 : \mathsf{code}(\Gamma)$ for any ψ and Γ must begin as follows:

$$\dfrac{\dfrac{\dfrac{\psi \vdash \mathsf{bar} : \mathsf{code}(\Gamma)}{\psi; \Gamma' \vdash \mathsf{bar} : \mathsf{code}(\Gamma)}\ \text{S-VAL}}{\psi \vdash r_1 := \mathsf{bar} : \Gamma' \to \Gamma}\ \text{S-MOV} \qquad \dfrac{\psi; \Gamma \vdash r_1 : \mathsf{code}(\Gamma)}{\psi \vdash \mathsf{jump}\ r_1 : \mathsf{code}(\Gamma)}\ \text{S-JUMP}}{\psi \vdash r_1 := \mathsf{bar}; \mathsf{jump}\ r_1 : \mathsf{Code}(\Gamma)}\ \text{S-SEQ}$$

But then $\Gamma = \Gamma'[(1, \Gamma)]$ and so in particular $\Gamma[(1, \mathsf{code}(\Gamma))]$. This kind of self-reference is problematic, in that there is no register valuation Γ with this behaivour, since register valuations are total functions.

One way to see the problem is to attempt to write such a register valuation as a set of pairs:

$$\Gamma = \{(1, \{(1, \ldots$$

Polymorphic types allow us to avoid this problem. In particular, let:

$$\Gamma = \{(1, \forall\alpha.\mathsf{code}(\{(1, \alpha)\}))\} \qquad \psi = \{(\mathsf{bar}, \forall\alpha.\mathsf{code}(\{1, \alpha\}))\}$$

Notice that we have:

$$\mathsf{code}(\Gamma) = \mathsf{code}(\{(1, \forall\alpha.\mathsf{code}(\{(1, \alpha)\}))\}) = \mathsf{code}(\{1, \alpha\})[\forall\alpha.\mathsf{code}(\{1, \alpha\}))/\alpha]$$

and so using the instantiation rule for our polymorphic types we have:

$$\dfrac{\dfrac{\dfrac{\mathsf{bar} \in \mathsf{dom}(\psi)}{\psi; \Gamma \vdash \mathsf{bar} : \forall\alpha.\mathsf{code}(\{(1, \alpha)\})}}{\psi; \Gamma \vdash \mathsf{bar} : \forall\alpha.\mathsf{code}(\{(1, \alpha)\})}}{\psi \vdash r_1 := \mathsf{bar} : \Gamma \to \Gamma} \qquad \dfrac{\dfrac{1 \leq k}{\psi; \Gamma \vdash r_1 : \forall\alpha.\mathsf{code}(\{1, \alpha\})}}{\psi; \Gamma \vdash r_1 : \mathsf{code}(\Gamma)}}{\psi \vdash \mathsf{jump}\ r_1 : \mathsf{code}(\Gamma)}$$
$$\psi \vdash r_1 := \mathsf{bar}; \mathsf{jump}\ r_1 : \mathsf{code}(\Gamma)$$

This explains why $\Gamma(4)$ has a polymorphic type in our example above, which we continue presently. Polymorphic types are also necessary to type some programs involving jumps to some location from multiple distinct locations.

Continued Example

We pick up our example again, showing that with the Γ,ψ, and H as before, we can derive:

$$\psi \vdash H(\mathsf{done}) : \mathsf{code}(\Gamma)$$

That is:

$$\psi \vdash \mathsf{jump}\ r_4 : \mathsf{code}(\Gamma)$$

With what we have just learned about polymorphic types, it is enough to notice:

$$\begin{aligned}&\mathsf{code}(\Gamma) = \mathsf{code}\{(1,\mathsf{int}),(2,\mathsf{int}),(3,\mathsf{int}),(4,\forall\alpha.\mathsf{code}\{(1,\mathsf{int}),(2,\mathsf{int}),(3,\mathsf{int}),(4,\alpha)\})\}\\ &= \mathsf{code}\{(1,\mathsf{int}),(2,\mathsf{int}),(3,\mathsf{int}),(4,\alpha)\}[\forall\alpha.\mathsf{code}\{(1,\mathsf{int}),(2,\mathsf{int}),(3,\mathsf{int}),(4,\alpha)\}/\alpha]\end{aligned}$$

Since then we have:

$$\dfrac{\dfrac{\dfrac{4 \le k}{\psi;\Gamma \vdash r_4 : \forall\alpha.\mathsf{code}\{(1,\mathsf{int}),(2,\mathsf{int}),(3,\mathsf{int}),(4,\alpha)\}}\ \text{S-REG}}{\psi;\Gamma \vdash r_4 : \mathsf{code}(\Gamma)}\ \text{S-INST}}{\psi \vdash \mathsf{jump}\ r_4 : \mathsf{code}(\Gamma)}\ \text{S-JUMP}$$

as required. Next, we derive:

$$\psi \vdash H(\mathsf{prod}) : \mathsf{code}(\Gamma)$$

that is:

$$\psi \vdash r_3 := 0; \mathsf{jump}\ \mathsf{loop} : \mathsf{code}(\Gamma)$$

This is straightforward. We have:

$$\dfrac{\dfrac{\dfrac{0 \in \mathbb{Z}}{\psi \vdash 0 : \mathsf{int}}}{\psi;\Gamma \vdash 0 : \mathsf{int}}}{\psi \vdash r_3 := 0 : \Gamma \to \Gamma} \qquad \dfrac{\dfrac{\dfrac{\mathsf{loop} \in \mathsf{dom}(\psi)}{\psi \vdash \mathsf{loop} : \mathsf{code}(\Gamma)}}{\psi;\Gamma \vdash \mathsf{loop} : \mathsf{code}(\Gamma)}}{\psi \vdash \mathsf{jump}\ \mathsf{loop} : \mathsf{code}(\Gamma)}$$

$$\dfrac{\psi \vdash r_3 := 0 : \Gamma \to \Gamma \qquad \psi \vdash \mathsf{jump}\ \mathsf{loop} : \mathsf{code}(\Gamma)}{\psi \vdash r_3 := 0; \mathsf{jump}\ \mathsf{loop} : \mathsf{code}(\Gamma)}$$

Notice that we have now shown $\vdash H : \psi$. Let R_0 be the starting register valuation from our example of program execution using the abstract machine with, say $R_0(3) = 56$ so that the function $R_0 : \{1,\ldots 4\} \to \mathcal{O}$ is fully specified. That is:

$$R_0 = \{(1,2),(2,2),(3,56),(4,\mathsf{exit})\}$$

It is easy to check that $\psi \vdash R_0 : \Gamma$ and $\psi \vdash \mathsf{jump}\ \mathsf{prod} : \mathsf{code}(\Gamma)$. Together, these facts give $\vdash (H, R_0, \mathsf{jump}\ \mathsf{prod})$. That is, our example machine state from last time is well-typed.

Exercises

- Using your program that computes the factorial function from last week's exercises, show that the machine state (H, R, I) which computes 3! when executed using the operational semantics is well-typed, in the sense that we may derive $\vdash (H, R, I)$ using the typing rules for TAL-0.
- Consider your program from last week's exercises that "gets stuck". Is it well-typed?

References

[1] Pierce, B.C. *Advanced Topics in Types and Programming Languages*. MIT Press, 2004.

$\boxed{\psi \vdash v : \tau}$ Value typing judgements:

$$\frac{n \in \mathbb{Z}}{\psi \vdash n : \mathsf{int}}\ \text{S-INT} \qquad \frac{l \in \mathsf{dom}(\psi)}{\psi \vdash l : \psi(l)}\ \text{S-LAB}$$

$\boxed{\psi; \Gamma \vdash v : \tau}$ Operand typing judgements:

$$\frac{i \leq k}{\psi; \Gamma \vdash r_i : \Gamma(i)}\ \text{S-REG} \qquad \frac{\psi \vdash v : \tau}{\psi; \Gamma \vdash v : \tau}\ \text{S-VAL} \qquad \frac{\psi; \Gamma \vdash v : \forall \alpha.\tau}{\psi; \Gamma \vdash v : \tau[\tau'/\alpha]}\ \text{S-INST}$$

$\boxed{\psi \vdash \iota : \Gamma_1 \to \Gamma_2}$ Instruction typing judgements:

$$\frac{\psi; \Gamma \vdash v : \tau}{\psi \vdash r_i := v : \Gamma \to \Gamma[(i, \tau)]}\ \text{S-MOVE} \qquad \frac{\psi; \Gamma \vdash r_j : \mathsf{int} \qquad \psi; \Gamma \vdash v : \mathsf{int}}{\psi \vdash r_i := r_j + v : \Gamma \to \Gamma[(r_i : \mathsf{int})]}\ \text{S-ADD}$$

$$\frac{\psi; \Gamma \vdash r_i : \mathsf{int} \qquad \psi; \Gamma \vdash v : \mathsf{code}(\Gamma)}{\psi \vdash \mathsf{if}\ r_i\ \mathsf{jump}\ v : \Gamma \to \Gamma}\ \text{S-COND}$$

$\boxed{\psi \vdash I : \tau}$ Instruction sequence typing judgements:

$$\frac{\psi; \Gamma \vdash v : \mathsf{code}(\Gamma)}{\psi \vdash \mathsf{jump}\ v : \mathsf{code}(\Gamma)}\ \text{S-JUMP} \qquad \frac{\psi \vdash I : \tau}{\psi \vdash \forall \alpha.\tau}\ \text{S-GEN}$$

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

$\boxed{\psi \vdash R : \Gamma}$ Register valuation typing judgements:

$$\frac{\forall i \in \{1, \ldots, k\}.\psi \vdash R(i) \vdash \Gamma(i)}{\psi \vdash R : \Gamma}\ \text{S-REGVAL}$$

$\boxed{\vdash H : \psi}$ Heap typing judgements:

$$\frac{\forall l \in \mathsf{dom}(\psi).\psi \vdash H(l) : \psi(l) \qquad \forall l \in \mathsf{dom}(\psi).\mathsf{FV}(\psi(l)) = \emptyset}{\vdash H : \psi}\ \text{S-HEAP}$$

$\boxed{\vdash (H, R, I)}$ Machine state typing judgements:

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

Figure 2: Typing rules for TAL-0