Syntax and Operational Semantics of TAL-0

Software Security

Spring 2025

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

Syntax

The syntax of TAL-0 is parameterised over a natural number $k \in \mathbb{N}$ (the number of registers) and a set $\mathcal{L}$ of *labels*, and is given in Figure 1.

Registers r as in:

$$r ::= r_1 \mid r_2 \mid \ldots \mid r_k$$

Operands v as in:

$$\begin{aligned} v ::=\ & n \in \mathbb{Z} && \text{(integer literals)} \\ \mid\ & l \in \mathcal{L} && \text{(labels)} \\ \mid\ & r && \text{(registers)} \end{aligned}$$

Instructions ι as in:

$$\begin{aligned} \iota ::=\ & r_d := v && \text{(move)} \\ \mid\ & r_d := r_s + v && \text{(add)} \\ \mid\ & \textsf{if } r \textsf{ jump } v && \text{(conditional jump)} \end{aligned}$$

Instruction sequences I as in:

$$\begin{aligned} I ::=\ & \textsf{jump } v && \text{(jump)} \\ \mid\ & \iota \,;\, I && \text{(sequence)} \end{aligned}$$

Figure 1: The Syntax of the TAL-0 Langauge

Let $\mathcal{O}$ be the set of all operands v in TAL-0, and let $\mathcal{S}$ be the set of all instruction sequences I. For example:

$$3 \in \mathcal{O} \qquad \mathsf{home} \in \mathcal{L} \Rightarrow \mathsf{home} \in \mathcal{O} \qquad x \leq k \Rightarrow r_x \in \mathcal{O}$$

$$v \in \mathcal{O} \Rightarrow \mathsf{jump}\ v \in \mathcal{S} \qquad x, y, v \in \mathcal{O} \Rightarrow r_x := r_y + v; \mathsf{jump}\ v \in \mathcal{S}$$

$$\mathsf{jump}\ 3 \in \mathsf{S}$$

Operational Semantics

TAL-0 programs are evaluated with respect to a *register valuation*, which is a function:

$$R : \{1, \ldots, k\} \to \mathcal{O}$$

together with a *heap*, which is a (finitely-supported) partial function:

$$H : \mathcal{L} \to \mathcal{S}$$

Every heap can be written as a (finite) set of pairs (l, I) with $l \in \mathcal{L}$ and $I \in \mathcal{S}$, subject to the restriction that each $l \in \mathcal{L}$ appears in at most one such pair. For example, if $\mathsf{one}, \mathsf{two}, \mathsf{three} \in \mathcal{L}$ and we define:

$$H = \{(\mathsf{one}, \mathsf{jump}\ 2), (\mathsf{two}, \mathsf{jump}\ 3), (\mathsf{three}, r_1 := r_2 + 1; \mathsf{jump}\ 4)\}$$

then we have:

$$H(\mathsf{one}) = \mathsf{jump2} \qquad H(\mathsf{two}) = \mathsf{jump3} \qquad H(\mathsf{three}) = r_1 := r_2 + 1; \mathsf{jump}\ 4,$$

with $H(l)$ undefined for any other $l \in \mathcal{L}$.

If H is a finitely supported partial function then for any $a \in \mathcal{L}$ and $X \in \mathcal{S}$ we may define another heap $H[(a, X)]$ as in:

$$H[(a, X)](l) = \begin{cases} X & \text{if a = l} \\ H(l) & \text{otherwise} \end{cases}$$

The operational semantics of TAL-0 is given by an abstract machine. A *state* of the abstract machine is a 3-tuple (H, R, I) where:

- $H : \mathcal{L} \to \mathcal{S}$ is a heap
- $R : \{1, \ldots, k\} \to \mathcal{O}$ is a register valuation
- $I \in \mathcal{S}$ is an instruction sequence

We define a function $\widehat{R} : \mathcal{O} \to \mathcal{O}$ as in:

$$\widehat{R}(v) = \begin{cases} R(i) & \text{if } v = r_i \\ n & \text{if } v = n \in \mathbb{Z} \\ l & \text{if } v = l \in \mathcal{L} \end{cases}$$

and we define a partial function $\widehat{H} : \mathcal{O} \to \mathcal{S}$ as in:

$$\widehat{H}(v) = \begin{cases} H(v) & \text{if } v = l \in \mathcal{L} \\ \uparrow & \text{otherwise} \end{cases}$$

Now the transition rules for our abstract machine are as in Figure 2.

$$\frac{\widehat{H}(\widehat{R}(v)) = I}{(H, R, \mathsf{jump}\ v) \to (H, R, I)}\ \text{JUMP}$$

$$\frac{}{(H, R, r_d := v; I) \to (H, R[(r_d, \widehat{R}(v))], I)}\ \text{MOVE}$$

$$\frac{R(r_s) = n_1 \in \mathbb{Z} \qquad \widehat{R}(v) = n_2 \in \mathbb{Z}}{(H, R, r_d := r_s + v; I) \to (H, R[(r_d, n_1 + n_2)], I)}\ \text{ADD}$$

$$\frac{R(r) = 0 \qquad \widehat{H}(\widehat{R}(v)) = J}{(H, R, \mathsf{if}\ r\ \mathsf{jump}\ v; I) \to (H, R, J)}\ \text{COND-1}$$

$$\frac{R(r) \neq 0}{(H, R, \mathsf{if}\ r\ \mathsf{jump}\ v; I) \to (H, R, I)}\ \text{COND-2}$$

Figure 2: Transition rules for the TAL-0 abstract machine

It is convenient to specify heaps as in:

```
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
```

where, ignoring the comments for a moment, this corresponds to the heap:

$\{(\mathsf{prod}, r_3 := 0; \mathsf{jump\ loop})$
$,(\mathsf{loop}, \mathsf{if}\ r_1\ \mathsf{jump\ done}; r_3 := r_2 + r_3; r_1 := r_1 + (-1); \mathsf{jump\ loop})$
$,(\mathsf{done}, \mathsf{jump}\ r4)\}$

Now, let H be the above heap, and suppose that $R_0(r_1) = 2$, $R_0(r_2) = 2$, and $R_0(r_4) = \mathsf{exit} \in \mathcal{L}$. Then we have:

$$
\begin{aligned}
&(H, R_0, \mathsf{jump\ prod})\\
\rightarrow&(H, R_0, r_3 := 0, \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 0)], \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 0)], \mathsf{if}\ r_1\ \mathsf{jump\ done}; r_3 := r_2 + r_3; r_1 := r_1 + (-1); \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 0)], r_3 := r_2 + r_3; r_1 + (-1); \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 2)], r_1 + (-1); \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 2), (r_1, 1)], \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 2), (r_1, 1)], \mathsf{if}\ r_1\ \mathsf{jump\ done}; r_3 := r_2 + r_3; r_1 := r_1 + (-1); \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 2), (r_1, 1)], r_3 := r_2 + r_3; r_1 := r_1 + (-1); \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 4), (r_1, 1)], r_1 := r_1 + (-1); \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 4), (r_1, 0)], \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 4), (r_1, 0)], \mathsf{if}\ r_1\ \mathsf{jump\ done}; r_3 := r_2 + r_3; r_1 := r_1 + (-1); \mathsf{jump\ loop})\\
\rightarrow&(H, R_0[(r_3, 4), (r_1, 0)], \mathsf{jump}\ r_4)
\end{aligned}
$$

In this way, the program encoded by our heap allows us to multiply the value stored in r_1 by the value stored in r_2, with the result ending up in r_3. Once finished, our program jumps to the label $\mathsf{exit} \in \mathcal{L}$ which we assume is stored in r_4.

Exercises

- Write a program (specified as a heap, as in the example above) that computes the factorial $n!$ of a positive integer $0 < n \in \mathbb{Z}$.
- Test your program by showing that it successfully computes 3! when run using the abstract machine. This should look something like computation above, in which we show that our program successfully computes the product of 2 and 2.
- Write a program that causes the machine to "get stuck", in the sense that the antecedent of a the relevant transition rule of the abstract machine is not satisfied.

References

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