Example

$$\mathsf{l} : \underbrace{\mathsf{r1} := \mathsf{l}; \mathsf{r2} := \mathsf{l'};\ \mathsf{jump}\ \mathsf{r2}}_{I} \qquad \mathsf{l'} : \underbrace{\mathsf{jump}\ \mathsf{r1}}_{I'}$$

We have the heap $H = \{\mathsf{l} \mapsto I, \mathsf{l'} \mapsto I'\}$.

Define heap type $\Psi = \left\{ \begin{array}{l} \mathsf{l} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\}, \\ \mathsf{l'} : \mathsf{Code}\{\mathsf{r1} : \Psi(\mathsf{l}), \mathsf{r2} : \mathsf{Top}\} \end{array} \right\}$

Define register file types

$$\Gamma_1 = \{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\}$$
$$\Gamma_2 = \{\mathsf{r1} : \Psi(\mathsf{l}), \mathsf{r2} : \mathsf{Top}\}$$
$$\Gamma_3 = \{\mathsf{r1} : \Psi(\mathsf{l}), \mathsf{r2} : \Psi(\mathsf{l'})\}$$

claim 1: $\Psi \vdash I : \mathsf{Code}(\Gamma_1)$

claim 1: $\Psi \vdash I : \mathsf{Code}(\Gamma_1)$

$$\dfrac{\dfrac{\mathsf{l} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\} \in \Psi}{\Psi, \Gamma_1 \vdash \mathsf{l} : \Psi(\mathsf{l})}\ \text{(T-Lab)}}{\Psi \vdash \mathsf{r1} := \mathsf{l} : \Gamma_1 \rightarrow \Gamma_2}\ \text{(T-Mov)}$$

claim 1: $\Psi \vdash I : \mathsf{Code}(\Gamma_1)$

$$\dfrac{\dfrac{\mathsf{l} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\} \in \Psi}{\Psi, \Gamma_1 \vdash \mathsf{l} : \Psi(\mathsf{l})}\,(\text{T-Lab})}{\Psi \vdash \mathsf{r1} := \mathsf{l} : \Gamma_1 \longrightarrow \Gamma_2}\,(\text{T-Mov})$$

$$\begin{array}{c} \vdots \\ \Psi \vdash \mathsf{r2} := \mathsf{l}' : \Gamma_2 \longrightarrow \Gamma_3 \end{array}$$

claim 1: $\Psi \vdash I : \mathsf{Code}(\Gamma_1)$

$$\dfrac{\dfrac{\mathsf{l} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\} \in \Psi}{\Psi, \Gamma_1 \vdash \mathsf{l} : \Psi(\mathsf{l})}\ (\text{T-Lab})}{\Psi \vdash \mathsf{r1} := \mathsf{l}\ : \Gamma_1 \to \Gamma_2}\ (\text{T-Mov}) \qquad \begin{array}{c}\vdots \\ \Psi \vdash \mathsf{r2} := \mathsf{l}'\ : \Gamma_2 \to \Gamma_3\end{array}$$

$$\dfrac{\dfrac{\Psi, \Gamma_3 \vdash \mathsf{r2} : \Psi(\mathsf{l}') \quad \mathsf{Code}(\Gamma_3) \sqsubseteq \Psi(\mathsf{l}')}{\Psi, \Gamma_3 \vdash \mathsf{r2} : \mathsf{Code}(\Gamma_3)}\ (\text{T-Sub})}{\Psi \vdash\ \mathsf{jump}\ \mathsf{r2} : \mathsf{Code}(\Gamma_3)}\ (\text{T-Jump})$$

$$\begin{aligned} \mathsf{Code}(\Gamma_3) \quad &= \mathsf{Code}\{\mathsf{r1} : \Psi(\mathsf{l}), \quad \mathsf{r2} : \Psi(\mathsf{l}')\} \\ \sqsubseteq \Psi(\mathsf{l}') \quad &= \mathsf{Code}\{\mathsf{r1} : \Psi(\mathsf{l}), \quad \mathsf{r2} : \mathsf{Top}\} \end{aligned}$$

because $\Psi(\mathsf{l}) \sqsubseteq_1 \Psi(\mathsf{l})$ and $\Psi(\mathsf{l}') \sqsubseteq_1 \mathsf{Top}$.

$$\dfrac{\begin{array}{c}\vdots\\ \Psi \vdash \mathsf{r1} := \mathsf{l} : \Gamma_1 \to \Gamma_2\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Psi : \mathsf{r2} := \mathsf{l}' : \Gamma_2 \longrightarrow \Gamma_3\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash \mathsf{jump}\ \mathsf{r2} : \mathsf{Code}(\Gamma_3)\end{array}}{\Psi \vdash \mathsf{r2} := \mathsf{l}';\ \mathsf{jump}\ \mathsf{r2} : \mathsf{Code}(\Gamma_2)}\ \text{(T-Seq)}}{\Psi \vdash I : \mathsf{Code}(\Gamma_1)}\ \text{(T-Seq)}$$

This proves claim 1.

$$\dfrac{\begin{array}{c}\vdots\\ \Psi \vdash \mathsf{r1} := \mathsf{l} : \Gamma_1 \to \Gamma_2\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Psi : \mathsf{r2} := \mathsf{l}' : \Gamma_2 \to \Gamma_3\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash \mathsf{jump}\ \mathsf{r2} : \mathsf{Code}(\Gamma_3)\end{array}}{\Psi \vdash \mathsf{r2} := \mathsf{l}';\ \mathsf{jump}\ \mathsf{r2} : \mathsf{Code}(\Gamma_2)}\,(\text{T-Seq})}{\Psi \vdash I : \mathsf{Code}(\Gamma_1)}\,(\text{T-Seq})$$

This proves claim 1.

claim 2: $\Psi \vdash I' : \mathsf{Code}(\Gamma_2)$

$$\dfrac{\dfrac{\Psi, \Gamma_2 \vdash \mathsf{r1} : \Psi(\mathsf{l}) \qquad \mathsf{Code}(\Gamma_2) \sqsubseteq \Psi(\mathsf{l})}{\Psi, \Gamma_2 \vdash \mathsf{r1} : \mathsf{Code}(\Gamma_2)}\,(\text{T-Sub})}{\Psi \vdash \mathsf{jump}\ \mathsf{r1} : \mathsf{Code}(\Gamma_2)}\,(\text{T-Jump})$$

Well typing of the heap

Recall that $H = \{\mathsf{l} \mapsto I, \mathsf{l}' \mapsto I'\}$ and $\Psi = \{\mathsf{l} : \mathsf{Code}(\Gamma_1), \mathsf{l}' : \mathsf{Code}(\Gamma_2)\}$.

$$\dfrac{\begin{array}{c}\vdots \\ \Psi \vdash I : \mathsf{Code}(\Gamma_1)\end{array} \qquad \begin{array}{c}\vdots \\ \Psi \vdash I' : \mathsf{Code}(\Gamma_2)\end{array}}{\vdash H : \Psi}\ \text{(T-Heap)}$$

Well typing of the heap

Recall that $H = \{\mathsf{l} \mapsto I, \mathsf{l}' \mapsto I'\}$ and $\Psi = \{\mathsf{l} : \mathsf{Code}(\Gamma_1), \mathsf{l}' : \mathsf{Code}(\Gamma_2)\}$.

$$\frac{\begin{array}{c}\vdots \\ \Psi \vdash I : \mathsf{Code}(\Gamma_1)\end{array} \qquad \begin{array}{c}\vdots \\ \Psi \vdash I' : \mathsf{Code}(\Gamma_2)\end{array}}{\vdash H : \Psi} \text{ (T-Heap)}$$

Well typing of register file

Suppose we want to start running the machine with the register file

$$R = \{\mathsf{r1} \mapsto \mathsf{0}, \mathsf{r2} \mapsto \mathsf{0}\}$$

Well typing of the heap

Recall that $H = \{\mathsf{l} \mapsto I, \mathsf{l}' \mapsto I'\}$ and $\Psi = \{\mathsf{l} : \mathsf{Code}(\Gamma_1), \mathsf{l}' : \mathsf{Code}(\Gamma_2)\}$.

$$\frac{\begin{array}{c}\vdots \\ \Psi \vdash I : \mathsf{Code}(\Gamma_1)\end{array} \qquad \begin{array}{c}\vdots \\ \Psi \vdash I' : \mathsf{Code}(\Gamma_2)\end{array}}{\vdash H : \Psi} \text{ (T-Heap)}$$

Well typing of register file

Suppose we want to start running the machine with the register file

$$R = \{\mathsf{r1} \mapsto \mathsf{0}, \mathsf{r2} \mapsto \mathsf{0}\}$$

Define register file type $\quad \Gamma = \{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}\}$

Well typing of the heap

Recall that $H = \{\mathsf{l} \mapsto I, \mathsf{l}' \mapsto I'\}$ and $\Psi = \{\mathsf{l} : \mathsf{Code}(\Gamma_1), \mathsf{l}' : \mathsf{Code}(\Gamma_2)\}$.

$$\dfrac{\begin{array}{c}\vdots \\ \Psi \vdash I : \mathsf{Code}(\Gamma_1)\end{array} \qquad \begin{array}{c}\vdots \\ \Psi \vdash I' : \mathsf{Code}(\Gamma_2)\end{array}}{\vdash H : \Psi}\ \text{(T-Heap)}$$

Well typing of register file

Suppose we want to start running the machine with the register file

$$R = \{\mathsf{r1} \mapsto \mathsf{0}, \mathsf{r2} \mapsto \mathsf{0}\}$$

Define register file type $\quad \Gamma = \{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}\}$

$$\dfrac{\dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\ \text{(T-Int)} \qquad \dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\ \text{(T-Int)}}{\Psi \vdash R : \Gamma}\ \text{(TRegfile)}$$

Suppose the initial instruction sequence we want to execute is I.

We have shown that $\Psi \vdash I : \mathsf{Code}(\Gamma_1)$ (claim 1).

Similarly we show $\Psi \vdash I : \mathsf{Code}(\Gamma)$.

Suppose the initial instruction sequence we want to execute is I.

We have shown that $\Psi \vdash I : \mathsf{Code}(\Gamma_1)$ (claim 1).

Similarly we show $\Psi \vdash I : \mathsf{Code}(\Gamma)$.

Finally, well typing of the machine

$$\dfrac{\begin{array}{c}\vdots \\ \vdash H : \Psi\end{array} \qquad \begin{array}{c}\vdots \\ \Psi \vdash R : \Gamma\end{array} \qquad \begin{array}{c}\vdots \\ \Psi \vdash I : \mathsf{Code}(\Gamma)\end{array}}{\vdash (H, R, I)} \text{ (T-Mach)}$$

Another example

```
prod :  r3 := 0;                loop :  if r1 jump done;
        jump loop                       r3 := r2 + r3;          done :  jump r4
                                        r1 := r1 + -1;
                                        jump loop
```

Another example

$$\mathsf{prod}: \quad \begin{array}{l} \mathsf{r3} := 0; \\ \mathsf{jump\ loop} \end{array} \qquad \mathsf{loop}: \quad \begin{array}{l} \mathsf{if\ r1\ jump\ done}; \\ \mathsf{r3} := \mathsf{r2} + \mathsf{r3}; \\ \mathsf{r1} := \mathsf{r1} + -1; \\ \mathsf{jump\ loop} \end{array} \qquad \mathsf{done}: \quad \mathsf{jump\ r4}$$

To complete the example we will have r4 contain the halt label.

$$\mathsf{halt}: \quad \mathsf{jump\ halt}$$

Another example

$$\mathsf{prod}: \quad \begin{array}{l} \mathsf{r3} := 0; \\ \mathsf{jump\ loop} \end{array} \qquad \mathsf{loop}: \quad \begin{array}{l} \mathsf{if\ r1\ jump\ done}; \\ \mathsf{r3} := \mathsf{r2} + \mathsf{r3}; \\ \mathsf{r1} := \mathsf{r1} + -1; \\ \mathsf{jump\ loop} \end{array} \qquad \mathsf{done}: \quad \mathsf{jump\ r4}$$

To complete the example we will have r4 contain the halt label.

$$\mathsf{halt}: \quad \mathsf{jump\ halt}$$

Name the instructions $\iota_1, \ldots, \iota_8$ and the instruction sequences I_1, I_2, I_3, I_4.

Let $\Gamma' = \{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}, \mathsf{r4} : \mathsf{Top}\}$

Let $\Gamma = \{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}, \mathsf{r4} : \mathsf{Code}(\Gamma')\}$

Let $H = \{\mathsf{prod} \mapsto I_1, \mathsf{loop} \mapsto I_2, \mathsf{done} \mapsto I_3, \mathsf{halt} \mapsto I_4\}$.

Let $\Psi = \{\mathsf{prod} : \mathsf{Code}(\Gamma), \mathsf{loop} : \mathsf{Code}(\Gamma), \mathsf{done} : \mathsf{Code}(\Gamma), \mathsf{halt} : \mathsf{Code}(\Gamma')\}$.

$$\cfrac{\cfrac{\cfrac{}{\Psi, \Gamma \vdash \mathsf{r3} : \mathsf{Int}}\ (\text{T-Reg}) \qquad \cfrac{}{\Psi, \Gamma \vdash \mathsf{0} : \mathsf{Int}}\ (\text{T-Int})}{\Psi \vdash \iota_1 : \Gamma \rightarrow \Gamma}\ (\text{T-Mov}) \qquad \cfrac{\cfrac{}{\Psi, \Gamma \vdash \mathsf{loop} : \mathsf{Code}(\Gamma)}\ (\text{T-Lab})}{\Psi \vdash \ \mathsf{jump\ loop} : \mathsf{Code}(\Gamma)}\ (\text{T-Jump})}{\Psi \vdash I_1 : \mathsf{Code}(\Gamma)}\ (\text{T-Seq})$$

$$
\dfrac{
  \dfrac{
    \dfrac{}{\Psi, \Gamma \vdash \mathsf{r3} : \mathsf{Int}}\ (\text{T-Reg})
    \quad
    \dfrac{}{\Psi, \Gamma \vdash \mathsf{0} : \mathsf{Int}}\ (\text{T-Int})
  }{\Psi \vdash \iota_1 : \Gamma \to \Gamma}\ (\text{T-Mov})
  \quad
  \dfrac{
    \dfrac{}{\Psi, \Gamma \vdash \mathsf{loop} : \mathsf{Code}(\Gamma)}\ (\text{T-Lab})
  }{\Psi \vdash \mathsf{jump\ loop} : \mathsf{Code}(\Gamma)}\ (\text{T-Jump})
}{\Psi \vdash I_1 : \mathsf{Code}(\Gamma)}\ (\text{T-Seq})
$$

Similarly, $\Psi \vdash I_2 : \mathsf{Code}(\Gamma)$.

$$\dfrac{\dfrac{}{\Psi, \Gamma \vdash \mathsf{r3} : \mathsf{Int}}\,(\text{T-Reg}) \quad \dfrac{}{\Psi, \Gamma \vdash \mathsf{0} : \mathsf{Int}}\,(\text{T-Int})}{\Psi \vdash \iota_1 : \Gamma \rightarrow \Gamma}\,(\text{T-Mov}) \quad \dfrac{\dfrac{}{\Psi, \Gamma \vdash \mathsf{loop} : \mathsf{Code}(\Gamma)}\,(\text{T-Lab})}{\Psi \vdash \mathsf{jump\ loop} : \mathsf{Code}(\Gamma)}\,(\text{T-Jump})$$

$$\dfrac{\cdots}{\Psi \vdash I_1 : \mathsf{Code}(\Gamma)}\,(\text{T-Seq})$$

Similarly, $\Psi \vdash I_2 : \mathsf{Code}(\Gamma)$.

$$\dfrac{\dfrac{\dfrac{}{\Psi, \Gamma \vdash \mathsf{r4} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}, \mathsf{r4} : \mathsf{Top}\}}\,(\text{T-Reg})}{\Psi, \Gamma \vdash \mathsf{r4} : \mathsf{Code}(\Gamma)}\,(\text{T-Sub})}{\Psi \vdash I_3 : \mathsf{Code}(\Gamma)}\,(\text{T-Jump})$$

$$
\dfrac{
  \dfrac{
    \dfrac{}{\Psi, \Gamma \vdash \mathsf{r3} : \mathsf{Int}}\ \text{(T-Reg)}
    \quad
    \dfrac{}{\Psi, \Gamma \vdash \mathsf{0} : \mathsf{Int}}\ \text{(T-Int)}
  }{\Psi \vdash \iota_1 : \Gamma \rightarrow \Gamma}\ \text{(T-Mov)}
  \qquad
  \dfrac{
    \dfrac{}{\Psi, \Gamma \vdash \mathsf{loop} : \mathsf{Code}(\Gamma)}\ \text{(T-Lab)}
  }{\Psi \vdash \ \mathsf{jump\ loop} : \mathsf{Code}(\Gamma)}\ \text{(T-Jump)}
}{\Psi \vdash I_1 : \mathsf{Code}(\Gamma)}\ \text{(T-Seq)}
$$

Similarly, $\Psi \vdash I_2 : \mathsf{Code}(\Gamma)$.

$$
\dfrac{
  \dfrac{
    \dfrac{}{\Psi, \Gamma \vdash \mathsf{r4} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}, \mathsf{r4} : \mathsf{Top}\}}\ \text{(T-Reg)}
  }{\Psi, \Gamma \vdash \mathsf{r4} : \mathsf{Code}(\Gamma)}\ \text{(T-Sub)}
}{\Psi \vdash I_3 : \mathsf{Code}(\Gamma)}\ \text{(T-Jump)}
$$

$$
\dfrac{
  \dfrac{}{\Psi, \Gamma' \vdash \mathsf{halt} : \mathsf{Code}(\Gamma')}\ \text{(T-Lab)}
}{\Psi \vdash I_4 : \mathsf{Code}(\Gamma')}\ \text{(T-Jump)}
$$

Hence we have well typing of the machine:

$$\dfrac{\begin{array}{cccc} \vdots & \vdots & \vdots & \vdots \\ I_1 : \mathsf{Code}(\Gamma) & I_2 : \mathsf{Code}(\Gamma) & I_3 : \mathsf{Code}(\Gamma) & I_4 : \mathsf{Code}(\Gamma') \end{array}}{\vdash H : \Psi} \text{(T-Heap)}$$

Hence we have well typing of the machine:

$$\dfrac{\dfrac{\vdots}{I_1 : \mathsf{Code}(\Gamma)} \quad \dfrac{\vdots}{I_2 : \mathsf{Code}(\Gamma)} \quad \dfrac{\vdots}{I_3 : \mathsf{Code}(\Gamma)} \quad \dfrac{\vdots}{I_4 : \mathsf{Code}(\Gamma')}}{\vdash H : \Psi}\ \text{(T-Heap)}$$

Define initial register file: $R = \{\mathsf{r1} \mapsto \mathsf{0}, \mathsf{r2} \mapsto \mathsf{0}, \mathsf{r3} \mapsto \mathsf{0}, \mathsf{r4} \mapsto \mathsf{halt}\}$

$$\dfrac{\dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\ \text{(T-Int)} \quad \ldots \quad \dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\ \text{(T-Int)} \quad \dfrac{}{\Psi, \_ \vdash \mathsf{halt} : \mathsf{Code}(\Gamma')}\ \text{(T-Int)}}{\Psi \vdash R : \Gamma}\ \text{(T-Regfile)}$$

Hence we have well typing of the machine:

$$\dfrac{\begin{array}{cccc} \vdots & \vdots & \vdots & \vdots \\ I_1 : \mathsf{Code}(\Gamma) & I_2 : \mathsf{Code}(\Gamma) & I_3 : \mathsf{Code}(\Gamma) & I_4 : \mathsf{Code}(\Gamma') \end{array}}{\vdash H : \Psi} \text{(T-Heap)}$$

Define initial register file: $R = \{\mathsf{r1} \mapsto \mathsf{0}, \mathsf{r2} \mapsto \mathsf{0}, \mathsf{r3} \mapsto \mathsf{0}, \mathsf{r4} \mapsto \mathsf{halt}\}$

$$\dfrac{\dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\text{(T-Int)} \quad \ldots \quad \dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\text{(T-Int)} \quad \dfrac{}{\Psi, \_ \vdash \mathsf{halt} : \mathsf{Code}(\Gamma')}\text{(T-Int)}}{\Psi \vdash R : \Gamma} \text{(T-Regfile)}$$

$$\dfrac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \Psi \vdash I_1 : \mathsf{Code}(\Gamma)}{\vdash (H, R, I_1)} \text{(T-Mach)}$$

Following instruction sequences are rejected by our type system.

$\mathsf{l1} : \mathsf{r1} := \mathsf{l2}; \mathsf{r3} := \mathsf{r2} + \mathsf{1}; \ldots$

$\mathsf{l3} : \mathsf{r1} := \mathsf{5};\ \mathsf{jump}\ \mathsf{r1}$

Following instruction sequences are rejected by our type system.

$\mathsf{l1} : \mathsf{r1} := \mathsf{l2}; \mathsf{r3} := \mathsf{r2} + \mathsf{1}; \ldots$

$\mathsf{l3} : \mathsf{r1} := \mathsf{5};\ \mathsf{jump}\ \mathsf{r1}$

- We haven't discussed how to check if a mchine is well typed. Alternative: use proof carrying code.

Following instruction sequences are rejected by our type system.

$\mathsf{l1} : \mathsf{r1} := \mathsf{l2}; \mathsf{r3} := \mathsf{r2} + 1; \ldots$

$\mathsf{l3} : \mathsf{r1} := 5;\ \mathsf{jump}\ \mathsf{r1}$

- We haven't discussed how to check if a mchine is well typed. Alternative: use proof carrying code.
- It is straightforward to translate TAL-0 programs to code for some real processor.

  If the TAL-0 program is well-typed then the translated code will behave properly.

Following instruction sequences are rejected by our type system.

$\mathsf{l1} : \mathsf{r1} := \mathsf{l2}; \mathsf{r3} := \mathsf{r2} + 1; \ldots$

$\mathsf{l3} : \mathsf{r1} := 5;\ \mathsf{jump}\ \mathsf{r1}$

- We haven't discussed how to check if a mchine is well typed. Alternative: use proof carrying code.
- It is straightforward to translate TAL-0 programs to code for some real processor.

  If the TAL-0 program is well-typed then the translated code will behave properly.

  . . . for that we of course need to prove type safety for TAL-0 . . .

Type safety for TAL-0

"well typed machines do not get stuck"

Progress: If $\vdash M$ then there is some M' such that $M \rightarrow M'$.

Preservation: If $\vdash M$ and $M \rightarrow M'$ then $\vdash M'$.

Type safety for TAL-0

"well typed machines do not get stuck"

Progress: If $\vdash M$ then there is some M' such that $M \rightarrow M'$.

Preservation: If $\vdash M$ and $M \rightarrow M'$ then $\vdash M'$.

Proof: by easy induction, case analysis…

Type safety for TAL-0

"well typed machines do not get stuck"

Progress: If $\vdash M$ then there is some M' such that $M \longrightarrow M'$.

Preservation: If $\vdash M$ and $M \longrightarrow M'$ then $\vdash M'$.

Proof: by easy induction, case analysis...

Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.

Type safety for TAL-0

"well typed machines do not get stuck"

Progress: If $\vdash M$ then there is some M' such that $M \to M'$.

Preservation: If $\vdash M$ and $M \to M'$ then $\vdash M'$.

Proof: by easy induction, case analysis…

Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.

A: The definitions are almost always wrong.

— Anonymous

An extension: TAL-1

We now also deal with memory safety.

Besides registers, we now have a potentially infinite memory, stack, pointers, and facilities for allocating space for data.

Already expressive enough for implementing simple programs from high level languages.

Memory safety: no reads to or writes from illegal memory locations.

Examples of new kinds of instructions

- $\mathsf{r1} := \mathsf{Mem}[\mathsf{r2} + \mathsf{4}]$

  r2 stores a pointer. We access the 4th location past the corresponding memory location. The value there is loaded in r1.

Examples of new kinds of instructions

- $\mathsf{r1} := \mathsf{Mem}[\mathsf{r2} + \mathsf{4}]$

  $\mathsf{r2}$ stores a pointer. We access the 4th location past the corresponding memory location. The value there is loaded in $\mathsf{r1}$.

- $\mathsf{Mem}[\mathsf{r2} + \mathsf{4}] := \mathsf{r1}$

  The reverse store operation.

Examples of new kinds of instructions

- $\mathsf{r1} := \mathsf{Mem}[\mathsf{r2} + \mathsf{4}]$

  r2 stores a pointer. We access the 4th location past the corresponding memory location. The value there is loaded in r1.

- $\mathsf{Mem}[\mathsf{r2} + \mathsf{4}] := \mathsf{r1}$

  The reverse store operation.

- $\mathsf{r1} := \mathsf{malloc}\ \mathsf{10}$

  allocate 10 words on the heap, and store corresponding pointer in r1.

Examples of new kinds of instructions

- $\mathsf{r1} := \mathsf{Mem}[\mathsf{r2} + 4]$

  r2 stores a pointer. We access the 4th location past the corresponding memory location. The value there is loaded in r1.

- $\mathsf{Mem}[\mathsf{r2} + 4] := \mathsf{r1}$

  The reverse store operation.

- $\mathsf{r1} := \mathsf{malloc}\ 10$

  allocate 10 words on the heap, and store corresponding pointer in r1.

- salloc 10

  allocate 10 words on the stack (and update stack pointer)

Example code.

```
r1 := malloc 5;           // allocate 5 words on heap
Mem[r1] := 10;            // store data in the first word
Mem[r1 + 1] := 20;        // store data in the first word
r2 := Mem[r1]             // load 10 into r2
```

Example code.

```
r1 := malloc 5;          // allocate 5 words on heap
Mem[r1] := 10;           // store data in the first word
Mem[r1 + 1] := 20;       // store data in the first word
r2 := Mem[r1]            // load 10 into r2
```

The following code has no well-defined behavior.

```
r1 := malloc 5;    // allocate 5 words on heap
r2 := malloc 5;    // allocate 5 words on heap
r3 := r1 + r2      // add the two pointers
```

Example code.

```
r1 := malloc 5;          // allocate 5 words on heap
Mem[r1] := 10;           // store data in the first word
Mem[r1 + 1] := 20;       // store data in the first word
r2 := Mem[r1]            // load 10 into r2
```

The following code has no well-defined behavior.

```
r1 := malloc 5;    // allocate 5 words on heap
r2 := malloc 5;    // allocate 5 words on heap
r3 := r1 + r2      // add the two pointers
```

Hence for type safety, we should at least have a different type for pointers.

Further the type system should distinguish between pointers to different types of data.

```
r1 := malloc 5;
Mem[r1] := 9;
r2 := Mem[r1]      // r1 stores a pointer, hence this is ok
 jump r2           // not ok, since r1 was a pointer to an integer
```

Hence the type-system should deal with types like ptr(Int), ptr(Code(Γ)), ptr(ptr(Int)), ...

```
                      // currently r1 : ptr(Code(...))
r3 := 5;
Mem[r1] := r3;        // now r1 : ptr(Int)
r4 := Mem[r1];        // r4 : Int
 jump r4              // of course ill-typed
```

Hence type of a register should be updated after a store through it.

Aliasing problem

Should the following be well typed?

```
                 // currently r1 : ptr(Code(...)), r2 : ptr(Code(...))
r3 := 5;
Mem[r1] := r3;   // now r1 : ptr(Int)
r4 := Mem[r2];   // load through r2. r4 :???
jump r4          // is this well-typed???
```

Aliasing problem

Should the following be well typed?

```
                    // currently r1 : ptr(Code(...)), r2 : ptr(Code(...))
 r3 := 5;
 Mem[r1] := r3;     // now r1 : ptr(Int)
 r4 := Mem[r2];     // load through r2. r4 :???
 jump r4            // is this well-typed???
```

Answer: depends on whether r1 and r2 point to the same location (aliasing).

Aliasing problem

Should the following be well typed?

```
                      // currently r1 : ptr(Code(...)), r2 : ptr(Code(...))
r3 := 5;
Mem[r1] := r3;        // now r1 : ptr(Int)
r4 := Mem[r2];        // load through r2. r4 :???
jump r4               // is this well-typed???
```

Answer: depends on whether r1 and r2 point to the same location (aliasing).

Problem: how should the type system keep track of aliasing?

Solution: have two kinds of memory locations.

Shared pointers: support aliasing. Different type of data cannot be written.

Unique pointers: no aliasing. Different kind of data can be written. Useful for allocating and initializing shared data structures, and for stack frames.

The instruction

$$\text{commit } r_d$$

declares a pointer to be shared, its type cannot change now.

The TAL-1 syntax: we make the following extensions to the TAL-0 syntax.

$r ::=$	registers
$\mathsf{r1} \mid \ldots \mid \mathsf{rk} \mid \mathsf{sp}$	ordinary registers and stack pointer
$\iota ::=$	instructions
$\ldots$	mov/add/if-jump
$r_d := \mathsf{Mem}[r_s + n]$	load from memory
$\mathsf{Mem}[r_d + n] := r_s$	store to memory
$r_d := \mathsf{malloc}\ n$	allocate n heap words
$\mathsf{commit}\ r_d$	make the pointer shared
$\mathsf{salloc}\ n$	allocate n stack words
$\mathsf{sfree}\ n$	free n stack words

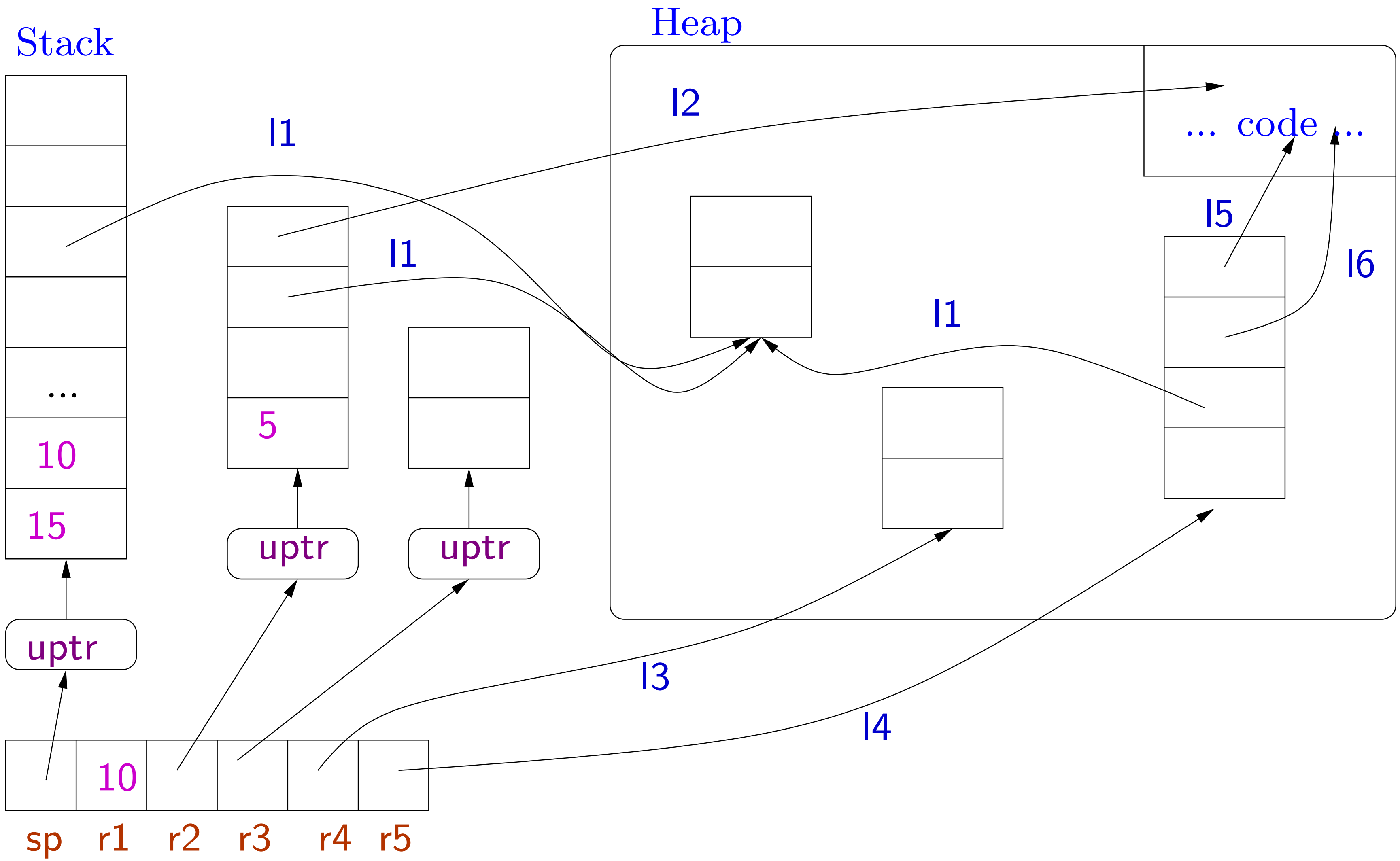
ν ::=	operands
r	registers
n	integers
l	code or shared data pointers
$\mathsf{uptr}(h)$	unique data pointers
h ::=	heap values
I	instruction sequences
$\langle \nu_1, \ldots, \nu_n \rangle$	tuples

Instruction sequences I are in TAL-0: list of instructions followed by a jump

Values are operands other than registers. Heaps map labels l to heap values h.

Register files and abstract machine states are defined as for TAL-0.

The TAL-1 abstract machine: Unique data values are not stored in the heap.



TAL-1 evaluation rules

We fix a constant MaxStack: the maximum allowed size of the stack.

All TAl-0 evaluation rules remain the same except the (E-Mov) rule.

This rule now needs to ensure that unique pointers are not copied.

$$\frac{\hat{R}(\nu) \neq \mathsf{uptr}(h)}{(H, R, r_d := \nu; I) \rightarrow (H, R \oplus \{r_d \mapsto \hat{R}(\nu)\}, I)} \text{ (E-Mov1)}$$

The $\hat{R}$ function is as for TAL-0. Further we have $\hat{R}(\mathsf{uptr}(h)) = \mathsf{uptr}(h)$.

If $\hat{R}(\nu)$ is $\mathsf{uptr}(h)$ then the machine gets stuck.

TAL-1 evaluation rules

We fix a constant MaxStack: the maximum allowed size of the stack.

All TAl-0 evaluation rules remain the same except the (E-Mov) rule.

This rule now needs to ensure that unique pointers are not copied.

$$\frac{\hat{R}(\nu) \neq \mathsf{uptr}(h)}{(H, R, r_d := \nu; I) \longrightarrow (H, R \oplus \{r_d \mapsto \hat{R}(\nu)\}, I)} \text{ (E-Mov1)}$$

The $\hat{R}$ function is as for TAL-0. Further we have $\hat{R}(\mathsf{uptr}(h)) = \mathsf{uptr}(h)$.

If $\hat{R}(\nu)$ is $\mathsf{uptr}(h)$ then the machine gets stuck.

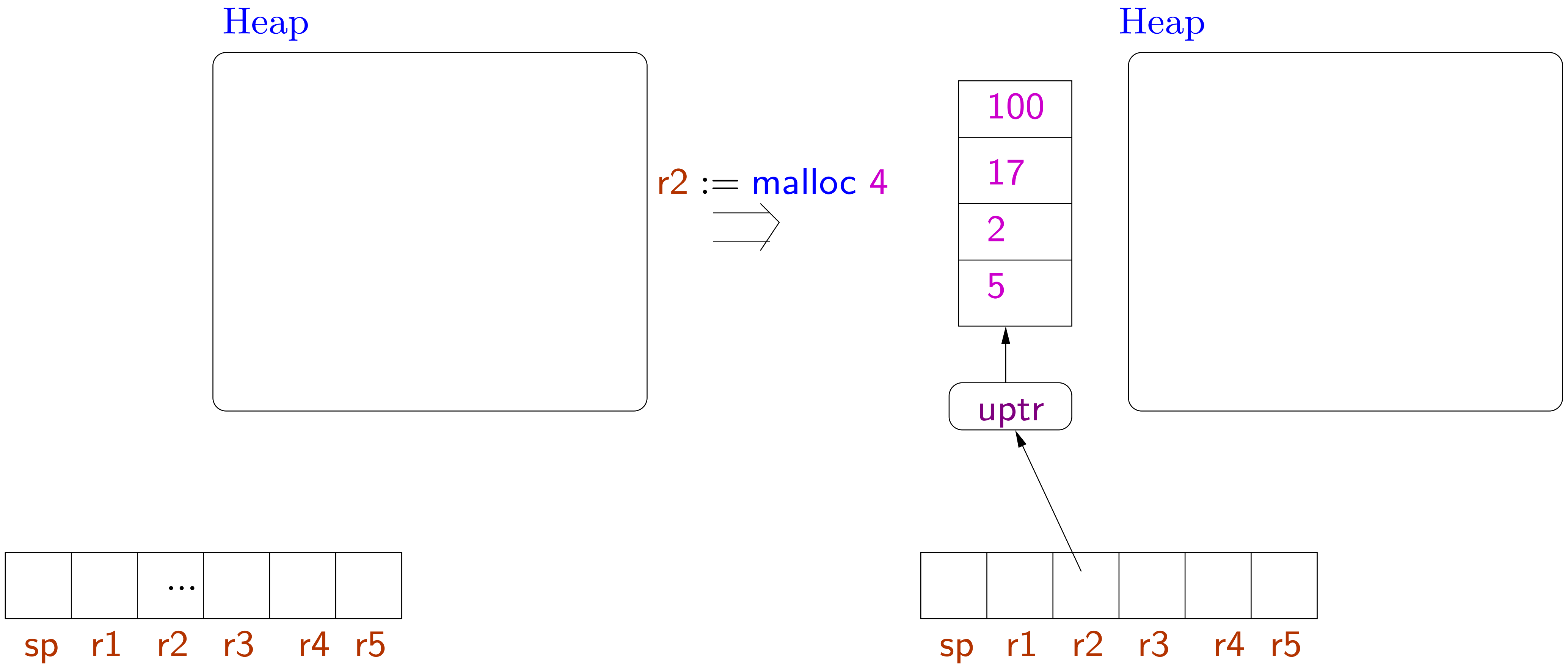
The other evaluation rules of TAL-0 are unmodified. We now add new rules for the new instructions ...

Allocation generates a unique pointer

$$(H, R, r_d := \mathsf{malloc}\ n; I) \to (H, R \oplus \{r_d \mapsto \mathsf{uptr}\langle m_1, \ldots, m_n\rangle\}, I) \qquad \text{(E-Malloc)}$$

- A unique pointer to a tuple of n words is created and stored in the destination register.
- The initial values in the words are arbitrary integers $m_1, \ldots, m_n$ (uninitialized values)
- Typically we would make the pointer shared once the words have been initialized.
- malloc instruction takes a constant as argument. Useful for implementing tuples, records, etc but not yet for variable sized arrays.

Allocation


Heap
r2 := malloc 4
...
sp r1 r2 r3 r4 r5
100
17
2
5
uptr
Heap
sp r1 r2 r3 r4 r5

Examples The following code will lead to stuck states.

- copying of unique pointers:

$$\ldots \mathsf{r1} := \mathsf{malloc}\ \mathsf{5}; \mathsf{r2} := \mathsf{r1}; \ldots$$

- using unique pointers in place of integers

$$\ldots \mathsf{r1} := \mathsf{malloc}\ \mathsf{5}; \mathsf{if}\ \mathsf{r1}\ \mathsf{jump}\ \mathsf{l}; \ldots$$

Declaring a pointer to be shared

$$\frac{r_d \neq \mathsf{sp} \quad R(r_d) = \mathsf{uptr}(h) \quad l \notin dom(H)}{(H, R, \mathsf{commit}\ r_d; I) \rightarrow (H \oplus \{l \mapsto h\}, R \oplus \{r_d \mapsto l\}, I)} \text{(E-Commit)}$$

- The stack is always a unique data value.
- commit moves the unique data in the heap (i.e. it is now considered shared data)
- A fresh label is associated with the data and is stored in the destination register.