Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\text{getfield } C.f.\tau} (\tau \cdot S, R)$$
$$\text{if } \tau' \sqsubseteq C$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\text{getfield } C.f.\tau} (\tau \cdot S, R)$$
$$\text{if } \tau' \sqsubseteq C$$

$$(\tau_1 \cdot \tau_2 \cdot S, R) \xrightarrow{\text{putfield } C.f.\tau} (S, R)$$
$$\text{if } \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq C$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\text{getfield } C.f.\tau} (\tau \cdot S, R)$$
$$\text{if } \tau' \sqsubseteq C$$

$$(\tau_1 \cdot \tau_2 \cdot S, R) \xrightarrow{\text{putfield } C.f.\tau} (S, R)$$
$$\text{if } \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq C$$

$$(\tau'_n \cdot \ldots \cdot \tau'_1 \cdot S, R) \xrightarrow{\text{invokestatic } C.m.\sigma} (\tau \cdot S, R)$$
$$\text{if } \sigma = \tau(\tau_1, \ldots, \tau_n), \tau'_i \sqsubseteq \tau_i \text{ for } 1 \leq i \leq n \text{ and } |\tau \cdot S| \leq M_{stack}$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\text{getfield } C.f.\tau} (\tau \cdot S, R)$$
$$\text{if } \tau' \sqsubseteq C$$

$$(\tau_1 \cdot \tau_2 \cdot S, R) \xrightarrow{\text{putfield } C.f.\tau} (S, R)$$
$$\text{if } \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq C$$

$$(\tau'_n \cdot \ldots \cdot \tau'_1 \cdot S, R) \xrightarrow{\text{invokestatic } C.m.\sigma} (\tau \cdot S, R)$$
$$\text{if } \sigma = \tau(\tau_1, \ldots, \tau_n), \tau'_i \sqsubseteq \tau_i \text{ for } 1 \leq i \leq n \text{ and } |\tau \cdot S| \leq M_{stack}$$

$$(\tau'_n \cdot \ldots \cdot \tau'_1 \cdot \tau' \cdot S, R) \xrightarrow{\text{invokevirtual } C.m.\sigma} (\tau \cdot S, R)$$
$$\text{if } \sigma = \tau(\tau_1, \ldots, \tau_n), \tau' \sqsubseteq C, \tau'_i \sqsubseteq \tau_i \text{ for } 1 \leq i \leq n \text{ and } |\tau \cdot S| \leq M_{stack}$$

Another example

```
public class testclass {
   public testclass () { }
   public Class testfunction (String s) {
      Class c = s.getClass();
      return c;
   }
}
```

```
public java.lang.Class testfunction(java.lang.String); 1 stack slots, 3 registers
    0: aload_1
    1: invokevirtual #2; //Method java/lang/Object.getClass:()Ljava/lang/Class;
    4: astore_2
    5: aload_2
    6: areturn
```

Our analysis on this example

```
public java.lang.Class testfunction(java.lang.String); 1 stack slots, 3 registers
                                    // stack, R(0), R(1), R(2)
                                    // ϵ, (testclass, String, ⊤)
    0:   aload_1                    // String, (testclass, String, ⊤)
    1:   invokevirtual #2;          // Class, (testclass, String, ⊤)
    4:   astore_2                   // ϵ, (testclass, String, Class)
    5:   aload_2                    // Class, (testclass, String, Class)
    6:   areturn
```

In case of several paths to a node, we need to compute least upper bounds $\sqcup$.

Comparison of abstract stacks:

$$T_1 \cdot \ldots \cdot T_n \sqsubseteq U_1 \cdot \ldots \cdot U_n \quad \text{iff} \quad T_i \sqsubseteq U_i \text{ for } 1 \leq i \leq n.$$

$$T_1 \cdot \ldots \cdot T_n \sqcup U_1 \cdot \ldots \cdot U_n = T_1 \sqcup U_1 \cdot \ldots \cdot T_n \sqcup U_n$$

Comparison of abstract register assignments:

$$R_1 \sqsubseteq R_2 \quad \text{iff} \quad R_1(i) \sqsubseteq R_2(i) \text{ for } 0 \leq i < M_{reg}.$$

$$(R_1 \sqcup R_2)(n) = R_1(n) \sqcup R_2(n)$$

Comparison of abstract states

$$(S_1, R_1) \sqsubseteq (S_2, R_2) \quad \text{iff} \quad S_1 \sqsubseteq S_2 \text{ and } R_1 \sqsubseteq R_2$$

$$(S_1, R_1) \sqcup (S_2, R_2) = (S_1 \sqcup S_2, R_1 \sqcup R_2)$$

Also $\bot \sqsubseteq (R, S)$ and $\bot \sqsubseteq (R, S) = (R, S)$.

Initial abstract state: (S_{start}, R_{start}) where $S_{start} = \epsilon$ is the empty stack and $R_{start}(0), \ldots, R_{start}(n-1)$ are the n arguments, and $R_{start}(i) = \top$ for $i \geq n$

If $\pi : pc_1 \rightarrow pc_2$ is a path (possibly with loops) from pc_1 to pc_2 with corresponding instruction sequence $I_1, \ldots, I_k$ and

$$(R_{i-1}, S_{i-1}) \xrightarrow{I_i} (S_i, R_i)$$

for $1 \leq i \leq n$ then we write $\pi : (S_0, R_0) \rightarrow (S_k, R_k)$.

For every valid location pc we define

Merge Over All Paths (MOP):

$$\mathcal{S}[pc] = \bigsqcup \{(S, R) \mid \pi : (S_{start}, R_{start}) \rightarrow (S, R)\}$$

Example

Suppose classes D and E are defined by extending class C, so that $D \sqcup E = C$.

```
                    // Int, (D, E)
10:  ifle  17       // ϵ, (D, E)
13:  aload_0        // D, (D, E)
14:  goto 18        // ϵ, (D, E)
17:  aload_1        // C, (D, E)
18:  areturn
```

(According to our notation, $C, (D, E)$ is the abstract state before the execution of the instruction at location 18.)

Another example

```
                    // ϵ, (Int, String)
9:   iload_0        // Int, (Int, String)
10:  ifle  17       // ϵ, (Int, String)
13:  iload_0        // Int, (Int, String)
14:  goto 18        // ϵ, (Int, String)
17:  aload_1        // ⊤, (Int, String)
18:  areturn
```

The bytecode verification fails because the return value is of unknown type.

```
public static int factorial(int); 2 stack slots, 2 registers
                          // ϵ, (Int, ⊤)
   0:   iconst_1          // Int, (Int, ⊤)
   1:   istore_1          // ϵ, (Int, Int)
   2:   iload_0           // Int, (Int, Int)
   3:   ifle 16           // ϵ, (Int, Int)
   6:   iload_1           // Int, (Int, Int)
   7:   iload_0           // Int · Int, (Int, Int)
   8:   imul              // Int, (Int, Int)
   9:   istore_1          // ϵ, (Int, Int)
   10:  iinc 0, −1        // ϵ, (Int, Int)
   13:  goto 2            // ϵ, (Int, Int)
   16:  iload_1           // Int, (Int, Int)
   17:  ireturn
```

Other issues to be tackled in the full Java bytecode language:

- initialization of objects
- exception handling

Typed Assembly Language (TAL)

Morrisett et al.

- A generic approach to safe compiled code.
- Based on the concept of type safety.
- Use type preserving compilation to transform type safe source code to type safe compiled code.
- Can be combined with the idea of proof carrying code.

A first language: TAL-0

Deals with control flow safety: no jumps to arbitrary machine addresses.

A first language: TAL-0

Deals with control flow safety: no jumps to arbitrary machine addresses.

Syntax of programs: We assume a fixed finite set of registers:

r ::=		registers
	r1 | ... | rk	
ν ::=		operands
	n	integer
	| l	label
	| r	register

A first language: TAL-0

Deals with control flow safety: no jumps to arbitrary machine addresses.

Syntax of programs: We assume a fixed finite set of registers:

r ::=		registers
	r1 | ... | rk	
ν ::=		operands
	n	integer
	| l	label
	| r	register

ι ::=		instructions
	$r_d := \nu$	
	| $r_d := r_s + \nu$	
	| if r jump ν	

A first language: TAL-0

Deals with control flow safety: no jumps to arbitrary machine addresses.

Syntax of programs: We assume a fixed finite set of registers:

$$
\begin{array}{rll}
r ::= & & \text{registers} \\
 & \mathsf{r1} \mid \ldots \mid \mathsf{rk} & \\
\nu ::= & & \text{operands} \\
 & n & \text{integer} \\
 & \mid l & \text{label} \\
 & \mid r & \text{register}
\end{array}
\qquad
\begin{array}{rll}
\iota ::= & & \text{instructions} \\
 & r_d := \nu & \\
 & \mid r_d := r_s + \nu & \\
 & \mid \mathsf{if}\ r\ \mathsf{jump}\ \nu & \\
I ::= & & \text{instruction sequences} \\
 & \mathsf{jump}\ \nu & \\
 & \mid \iota; I &
\end{array}
$$

A first language: TAL-0

Deals with control flow safety: no jumps to arbitrary machine addresses.

Syntax of programs: We assume a fixed finite set of registers:

r ::= registers

r1 | ... | rk

ν ::= operands

n integer

| l label

| r register

ι ::= instructions

$r_d := \nu$

| $r_d := r_s + \nu$

| if r jump ν

I ::= instruction sequences

jump ν

| $\iota; I$

Operands other than registers are called values (i.e. registers and labels).

• Instruction sequences have an unconditional jump at the end, and other instructions before.

• As yet, no infinite memory (except for code).

• Instruction sequences have an unconditional jump at the end, and other instructions before.

• As yet, no infinite memory (except for code).

An example for computing product: r4 contains the return address

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

The example has three instruction sequences, and a label corresponding to each of them.

Evaluation: the TAL-0 abstract machine

- the abstract machine contains the code and data.
- an evaluation step changes the state (code and data) of the abstract machine.

R ::= register files

$\{\mathsf{r1} \mapsto \nu_1, \ldots, \mathsf{rk} \mapsto \nu_k\}$ (each ν_i is a value)

h ::= heap values

I instruction sequences

H ::= heaps

$\{l_1 \mapsto h_1, \ldots l_m \mapsto h_m\}$

M ::= abstract machine states

(H, R, I) (I is the current instruction sequence being executed)

- A register file R maps each register r to some value (integer or label) $R(r)$.

- A register file R maps each register r to some value (integer or label) $R(r)$.
- A heap H is a partial map: H maps some labels l to heap values $H(l)$.

- A register file R maps each register r to some value (integer or label) $R(r)$.
- A heap H is a partial map: H maps some labels l to heap values $H(l)$.

The previous example has three instruction sequences

$I_1 = \mathsf{r3} := 0;\ \mathsf{jump\ loop}$

$I_2 = \mathsf{if\ r1\ jump\ done}; \mathsf{r3} := \mathsf{r2} + \mathsf{r3}; \mathsf{r1} := \mathsf{r1} + -1;\ \mathsf{jump\ loop}$

$I_3 =\ \mathsf{jump\ r4}$

We have the heap $H_0 = \{\mathsf{prod} \mapsto I_1, \mathsf{loop} \mapsto I_2, \mathsf{done} \mapsto I_3\}$.

- A register file R maps each register r to some value (integer or label) $R(r)$.
- A heap H is a partial map: H maps some labels l to heap values $H(l)$.

The previous example has three instruction sequences

$I_1 = \mathsf{r3} := 0;\ \mathsf{jump\ loop}$

$I_2 = \mathsf{if\ r1\ jump\ done}; \mathsf{r3} := \mathsf{r2} + \mathsf{r3}; \mathsf{r1} := \mathsf{r1} + -1;\ \mathsf{jump\ loop}$

$I_3 =\ \mathsf{jump\ r4}$

We have the heap $H_0 = \{\mathsf{prod} \mapsto I_1, \mathsf{loop} \mapsto I_2, \mathsf{done} \mapsto I_3\}$.

The starting state of the machine is supposed to be of the form

$$M_0 = (H_0, R_0, I_1)$$

where $R_0(\mathsf{r1}) = \mathsf{n}$ and $R_0(\mathsf{r2}) = \mathsf{m}$ are integers and $R_0(\mathsf{r4})$ is a label.

A possible execution sequence: ...

$$H_0,\quad \{r1 \mapsto 2,\quad r2 \mapsto 2,\quad r3 \mapsto 0,\quad r4 \mapsto \mathsf{I}\},\qquad \mathsf{r3} := 0;\ \text{jump loop}$$

$$H_0,\quad \{r1 \mapsto 2,\quad r2 \mapsto 2,\quad r3 \mapsto 0\quad r4 \mapsto \mathsf{I}\},\qquad \text{jump loop}$$

$$H_0,\quad \{r1 \mapsto 2,\quad r2 \mapsto 2,\quad r3 \mapsto 0\quad r4 \mapsto \mathsf{I}\},\qquad I_2$$

$$H_0,\quad \{r1 \mapsto 2,\quad r2 \mapsto 2,\quad r3 \mapsto 0\quad r4 \mapsto \mathsf{I}\},\qquad r3 := r2 + r3; \mathsf{r1} := \mathsf{r1} + -1;\ \text{jump loop}$$

$$H_0,\quad \{r1 \mapsto 2,\quad r2 \mapsto 2,\quad r3 \mapsto 2\quad r4 \mapsto \mathsf{I}\},\qquad \mathsf{r1} := \mathsf{r1} + -1;\ \text{jump loop}$$

$$H_0,\quad \{r1 \mapsto 1,\quad r2 \mapsto 2,\quad r3 \mapsto 2\quad r4 \mapsto \mathsf{I}\},\qquad \text{jump loop}$$

$$H_0,\quad \{r1 \mapsto 1,\quad r2 \mapsto 2,\quad r3 \mapsto 2\quad r4 \mapsto \mathsf{I}\},\qquad I_2$$

$$H_0,\quad \{r1 \mapsto 1,\quad r2 \mapsto 2,\quad r3 \mapsto 2\quad r4 \mapsto \mathsf{I}\},\qquad r3 := r2 + r3; \mathsf{r1} := \mathsf{r1} + -1;\ \text{jump loop}$$

$$H_0,\quad \{r1 \mapsto 1,\quad r2 \mapsto 2,\quad r3 \mapsto 4\quad r4 \mapsto \mathsf{I}\},\qquad \mathsf{r1} := \mathsf{r1} + -1;\ \text{jump loop}$$

$$H_0,\quad \{r1 \mapsto 0,\quad r2 \mapsto 2,\quad r3 \mapsto 4\quad r4 \mapsto \mathsf{I}\},\qquad \text{jump loop}$$

$$H_0,\quad \{r1 \mapsto 0,\quad r2 \mapsto 2,\quad r3 \mapsto 4\quad r4 \mapsto \mathsf{I}\},\qquad I_2$$

$$H_0,\quad \{r1 \mapsto 0,\quad r2 \mapsto 2,\quad r3 \mapsto 4\quad r4 \mapsto \mathsf{I}\},\qquad \text{jump } r4$$

As usual, we formalize this using evaluation rules.

As usual, we formalize this using evaluation rules.

$$\frac{H(\hat{R}(\nu)) = I}{(H, R, \text{ jump } \nu) \longrightarrow (H, R, I)} \text{ E-Jump)}$$

where the lookup function $\hat{R}$ returns the value corresponding to an operand:

$$\hat{R}(r) = R(r)$$
$$\hat{R}(n) = n$$
$$\hat{R}(l) = l$$

The JUMP instruction loads a new instruction sequence which should then be executed.

(The machine is stuck if $\hat{R}(\nu)$ is not a label.)

Otherwise, we consume one instruction from the current instruction sequence.

The MOV and ADD instructions modify the register file.

$$(H, R, r_d := \nu; I) \longrightarrow (H, R \oplus \{r_d \mapsto \hat{R}(\nu)\}, I) \qquad \text{(E-Mov)}$$

Otherwise, we consume one instruction from the current instruction sequence.

The MOV and ADD instructions modify the register file.

$$(H, R, r_d := \nu; I) \longrightarrow (H, R \oplus \{r_d \mapsto \hat{R}(\nu)\}, I) \qquad \text{(E-Mov)}$$

$$\frac{R(r_s) = n_1 \qquad \hat{R}(\nu) = n_2}{(H, R, r_d := r_s + \nu; I) \longrightarrow (H, R \oplus \{r_d \mapsto n_1 + n_2\}, I)} \text{(E-Add)}$$

(The machine is stuck in the second case if $R(r_s)$ or $\hat{R}(\nu)$ is not an integer.)

The conditional jump instruction either loads a new instruction sequence or just consumes one instruction.

$$\frac{R(r) = 0 \qquad H(\hat{R}(\nu)) = I'}{(H, R, \text{if } r \text{ jump } \nu; I) \longrightarrow (H, R, I')} \text{ (E-IfEq)}$$

The conditional jump instruction either loads a new instruction sequence or just consumes one instruction.

$$\frac{R(r) = 0 \qquad H(\hat{R}(\nu)) = I'}{(H, R, \text{if } r \text{ jump } \nu; I) \longrightarrow (H, R, I')} \text{ (E-IfEq)}$$

$$\frac{R(r) = n \qquad n \neq 0}{(H, R, \text{if } r \text{ jump } \nu; I) \longrightarrow (H, R, I)} \text{ (E-IfNeq)}$$

(The machine is stuck if $R(r)$ is not an integer or, in the first case, if $\hat{R}(\nu)$ is not a label.)

Consider the following simple code:

```
l :   r1 := 5;
      jump r1
```

Consider the following simple code:

$$\begin{array}{ll} \mathsf{l}: & \mathsf{r1} := 5; \\ & \mathsf{jump\ r1} \end{array}$$

Define instruction sequence $I = \mathsf{r1} := 5;\ \mathsf{jump\ r1}$ and heap $H = \{\mathsf{l} \mapsto I\}$.

Corresponding to the above code, starting with register file $R = \{\mathsf{r1} \mapsto 0\}$ we have the evaluation step

$$(H, \{\mathsf{r1} \mapsto 0\}, I) \longrightarrow (H, \{\mathsf{r1} \mapsto 5\},\ \mathsf{jump\ r1})$$

Consider the following simple code:

$$\begin{array}{ll} \mathsf{l}: & \mathsf{r1} := 5; \\ & \mathsf{jump\ r1} \end{array}$$

Define instruction sequence $I = \mathsf{r1} := 5;\ \mathsf{jump\ r1}$ and heap $H = \{\mathsf{l} \mapsto I\}$.

Corresponding to the above code, starting with register file $R = \{\mathsf{r1} \mapsto 0\}$ we have the evaluation step

$$(H, \{\mathsf{r1} \mapsto 0\}, I) \longrightarrow (H, \{\mathsf{r1} \mapsto 5\},\ \mathsf{jump\ r1})$$

The machine is now stuck: no further evaluation step is possible because r1 stores an integer instead of a label.

Consider the following simple code:

$$\begin{array}{ll} \mathsf{l}: & \mathsf{r1} := 5; \\ & \mathsf{jump\ r1} \end{array}$$

Define instruction sequence $I = \mathsf{r1} := 5;\ \mathsf{jump\ r1}$ and heap $H = \{\mathsf{l} \mapsto I\}$.

Corresponding to the above code, starting with register file $R = \{\mathsf{r1} \mapsto 0\}$ we have the evaluation step

$$(H, \{\mathsf{r1} \mapsto 0\}, I) \longrightarrow (H, \{\mathsf{r1} \mapsto 5\},\ \mathsf{jump\ r1})$$

The machine is now stuck: no further evaluation step is possible because r1 stores an integer instead of a label.

Hence to filter out such bad programs, we need to introduce typing rules.

Initial idea for a TAL-0 typing system: introduce two different types Int and Code for integers and labels.

In the previous example, we will start with the register file type $\Gamma = \{\mathsf{r1} : \mathsf{Int}\}$.

After the instruction $\mathsf{r1} = 5$ the register file type remains the same.

Then the second instruction jump r1 fails to type check because $\Gamma(\mathsf{r1})$ is Int instead of Code.

Hence the code is rejected, as desired.

Initial idea for a TAL-0 typing system: introduce two different types Int and Code for integers and labels.

In the previous example, we will start with the register file type $\Gamma = \{\mathsf{r1} : \mathsf{Int}\}$.

After the instruction $\mathsf{r1} = 5$ the register file type remains the same.

Then the second instruction jump r1 fails to type check because $\Gamma(\mathsf{r1})$ is Int instead of Code.

Hence the code is rejected, as desired.

Is this idea enough?

Consider the following code:

$$\begin{array}{ll} \mathsf{l}: & \mathsf{r1} := 5; \\ & \mathsf{r2} := \mathsf{l}'; \\ & \mathsf{jump\ r2} \end{array}$$

Label l' points to some other instruction sequence I'.

$I = \mathsf{r1} := 5;\ \mathsf{jump\ r1}$ and heap $H = \{\mathsf{l} : I, \mathsf{l}' \mapsto I'\}$.

Should the above code be well-typed? After the first two instructions, the register file type will be $\{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Code}\}$, as it should be.

Answer: depends on I'...

Consider the code

$$\mathsf{l}' : \quad \mathsf{jump\ r1};$$

Clearly the instruction sequence $I' = \mathsf{jump\ r1}$ expects a label in r1 instead of an integer.

Hence the code at l is not well-typed.

Solution:

With each instruction sequence, associate a register file type that is expected at the beginning of that instruction sequence.

Secondly, enrich the notion of types. Instead of having a simple type Code for labels, we have types of the form $\mathsf{Code}(\Gamma)$ where Γ is a register file type.

We further choose a type Top which is the super type of all types.

In the previous example, the instruction sequence I' will have type

$$\{\mathsf{r1} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\}\}$$

The instruction sequence I' expects r1 to contain label to some instruction sequence (I) which expects both registers to contain "anything".

The instruction sequence I has type $\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\}$.

After executing the first two instructions of I, the register file type becomes $\{r1 : \mathsf{Int}, \mathsf{r2} : \mathsf{Code}\{\ldots\}$.

Hence the jump instruction doesn't type check.

The TAL-0 type system

τ ::=		operand types
	Int	integers
	$\mathsf{Code}(\Gamma)$	labels
	Top	"any" type

The TAL-0 type system

τ ::= operand types

- Int integers
- $\mathsf{Code}(\Gamma)$ labels
- Top "any" type

Γ ::= register file types

$\{\mathsf{r1} : \tau_1, \ldots, \mathsf{rk} : \tau_k\}$

Ψ ::= heap types

$\{l_1 : \tau_1, \ldots, l_m : \tau_m\}$

The TAL-0 type system

$\tau ::=$	operand types	$\Gamma ::=$	register file types
Int	integers	$\{\mathsf{r1} : \tau_1, \ldots, \mathsf{rk} : \tau_k\}$	
$\mathsf{Code}(\Gamma)$	labels	$\Psi ::=$	heap types
Top	"any" type	$\{l_1 : \tau_1, \ldots, l_m : \tau_m\}$	

Typing of operands

The type judgment

$$\Psi, \Gamma \vdash \nu : \tau$$

means: under heap type Ψ and register file type Γ, the operand ν has type τ.

The TAL-0 type system

$\tau ::=$	operand types	$\Gamma ::=$	register file types
Int	integers	$\{\mathsf{r1} : \tau_1, \ldots, \mathsf{rk} : \tau_k\}$	
$\mathsf{Code}(\Gamma)$	labels	$\Psi ::=$	heap types
Top	"any" type	$\{l_1 : \tau_1, \ldots, l_m : \tau_m\}$	

Typing of operands

The type judgment

$$\Psi, \Gamma \vdash \nu : \tau$$

means: under heap type Ψ and register file type Γ, the operand ν has type τ.

$$\Psi, \Gamma \vdash n : \mathsf{Int} \quad \text{(T-Int)}$$

The TAL-0 type system

$\tau ::=$		operand types	$\Gamma ::=$		register file types
	Int	integers		$\{\mathsf{r1} : \tau_1, \ldots, \mathsf{rk} : \tau_k\}$	
	$\mathsf{Code}(\Gamma)$	labels	$\Psi ::=$		heap types
	Top	"any" type		$\{l_1 : \tau_1, \ldots, l_m : \tau_m\}$	

Typing of operands

The type judgment

$$\Psi, \Gamma \vdash \nu : \tau$$

means: under heap type Ψ and register file type Γ, the operand ν has type τ.

$$\Psi, \Gamma \vdash n : \mathsf{Int} \quad \text{(T-Int)} \qquad \frac{l : \tau \in \Psi}{\Psi, \Gamma \vdash l : \tau} \quad \text{(T-Lab)}$$

$$\Psi, \Gamma \vdash r : \Gamma(r) \qquad \text{(T-Reg)}$$

$$\Psi, \Gamma \vdash r : \Gamma(r) \qquad \text{(T-Reg)}$$

$$\frac{\Psi, \Gamma \vdash \nu : \tau \qquad \tau' \sqsubseteq \tau}{\Psi, \Gamma \vdash \nu : \tau'} \text{(T-Sub)}$$

$$\Psi, \Gamma \vdash r : \Gamma(r) \quad \text{(T-Reg)}$$

$$\frac{\Psi, \Gamma \vdash \nu : \tau \qquad \tau' \sqsubseteq \tau}{\Psi, \Gamma \vdash \nu : \tau'} \text{ (T-Sub)}$$

where

τ	$\sqsubseteq_1 \tau$	for every τ
τ	$\sqsubseteq_1 \mathsf{Top}$	for every τ
$\mathsf{Code}(\Gamma_1)$	$\sqsubseteq \mathsf{Code}(\Gamma_2)$	iff $\Gamma_1(r) \sqsubseteq_1 \Gamma_2(r)$ for every register r

Top represents "any" type, hence can be replaced by any type.

Typing of instructions

The type judgment

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

means: under heap type Ψ, the instruction ι modifies the register file type from Γ_1 to Γ_2.

Typing of instructions

The type judgment

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

means: under heap type Ψ, the instruction ι modifies the register file type from Γ_1 to Γ_2.

$$\frac{\Psi, \Gamma \vdash \nu : \tau}{\Psi \vdash r_d := \nu : \Gamma \to \Gamma \oplus \{r_d : \tau\}} \text{(T-Mov)}$$

Typing of instructions

The type judgment

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

means: under heap type Ψ, the instruction ι modifies the register file type from Γ_1 to Γ_2.

$$\frac{\Psi, \Gamma \vdash \nu : \tau}{\Psi \vdash r_d := \nu : \Gamma \to \Gamma \oplus \{r_d : \tau\}} \text{(T-Mov)}$$

$$\frac{\Psi, \Gamma \vdash r_s : \mathsf{Int} \qquad \Psi, \Gamma \vdash \nu : \mathsf{Int}}{\Psi \vdash r_d := r_s + \nu : \Gamma \to \Gamma \oplus \{r_d : \mathsf{Int}\}} \text{(T-Add)}$$

The mov and add instructions modify the type of the destination register.

$$\frac{\Psi, \Gamma \vdash r_s : \mathsf{Int} \qquad \Psi, \Gamma \vdash \nu : \mathsf{Code}(\Gamma)}{\Psi \vdash \mathsf{if}\ r_s\ \mathsf{jump}\ \nu : \Gamma \rightarrow \Gamma}\ (\text{T-If})$$

Both branches of the if instruction must have the same type.

If the if condition fails then the next instruction is executed with register file of type Γ.

If the if condition succeeds then the jump should be to some instruction sequence which expects register file type Γ.

Typing of instruction sequences

The type judgment

$$\Psi : I : \mathsf{Code}(\Gamma)$$

means: under heap type Ψ, the instruction sequence I expects the register file to have type Γ at the beginning.

Typing of instruction sequences

The type judgment

$$\Psi : I : \mathsf{Code}(\Gamma)$$

means: under heap type Ψ, the instruction sequence I expects the register file to have type Γ at the beginning.

$$\frac{\Psi, \Gamma \vdash \nu : \mathsf{Code}(\Gamma)}{\Psi \vdash \ \mathsf{jump}\ \nu : \mathsf{Code}(\Gamma)}\ (\text{T-Jump})$$

Typing of instruction sequences

The type judgment

$$\Psi : I : \mathsf{Code}(\Gamma)$$

means: under heap type Ψ, the instruction sequence I expects the register file to have type Γ at the beginning.

$$\frac{\Psi, \Gamma \vdash \nu : \mathsf{Code}(\Gamma)}{\Psi \vdash \ \mathsf{jump}\ \nu : \mathsf{Code}(\Gamma)}\ (\text{T-Jump})$$

$$\frac{\Psi \vdash \iota : \Gamma_1 \longrightarrow \Gamma_2 \qquad \Psi \vdash I : \mathsf{Code}(\Gamma_2)}{\Psi \vdash \iota; I : \mathsf{Code}(\Gamma_1)}\ (\text{T-Seq})$$

Typing of register files, heaps, and machine states

$$\frac{\Psi, \_ \vdash R(\mathsf{r1}) : \Gamma(\mathsf{r1}) \qquad \ldots \qquad \Psi, \_ \vdash R(\mathsf{rk}) : \Gamma(\mathsf{rk})}{\Psi \vdash R : \Gamma}\ \text{(T-Regfile)}$$

$\_$ means that the register file type is irrelevant here

Typing of register files, heaps, and machine states

$$\frac{\Psi, \_ \vdash R(\mathsf{r1}) : \Gamma(\mathsf{r1}) \qquad \ldots \qquad \Psi, \_ \vdash R(\mathsf{rk}) : \Gamma(\mathsf{rk})}{\Psi \vdash R : \Gamma} \text{ (T-Regfile)}$$

$\_$ means that the register file type is irrelevant here

$$\frac{\forall l \in dom(\Psi) \cdot \Psi \vdash H(l) : \Psi(l)}{\vdash H : \Psi} \text{ (T-Heap)}$$

$dom(\Psi)$ is the set of labels in the domain of Ψ

Typing of register files, heaps, and machine states

$$\frac{\Psi, \_ \vdash R(\mathsf{r1}) : \Gamma(\mathsf{r1}) \qquad \ldots \qquad \Psi, \_ \vdash R(\mathsf{rk}) : \Gamma(\mathsf{rk})}{\Psi \vdash R : \Gamma} \text{ (T-Regfile)}$$

$\_$ means that the register file type is irrelevant here

$$\frac{\forall l \in dom(\Psi) \cdot \Psi \vdash H(l) : \Psi(l)}{\vdash H : \Psi} \text{ (T-Heap)}$$

$dom(\Psi)$ is the set of labels in the domain of Ψ

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

The last judgment means that (H, R, I) is a well-typed machine.