Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\textsf{iconst}\ n} (\textsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\textsf{iconst}\ n} (\textsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

$$(\textsf{Int} \cdot \textsf{Int} \cdot S, R) \xrightarrow{\textsf{iadd}} (\textsf{Int} \cdot S, R)$$

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\text{iconst } n} (\mathsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

$$(\mathsf{Int} \cdot \mathsf{Int} \cdot S, R) \xrightarrow{\text{iadd}} (\mathsf{Int} \cdot S, R)$$

$$(S, R) \xrightarrow{\text{iload } n} (\mathsf{Int} \cdot S, R)$$

if $0 \leq n < M_{reg}$ and $R(n) = \mathsf{Int}$ and $|S| < M_{stack}$

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\text{iconst } n} (\mathsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

$$(\mathsf{Int} \cdot \mathsf{Int} \cdot S, R) \xrightarrow{\text{iadd}} (\mathsf{Int} \cdot S, R)$$

$$(S, R) \xrightarrow{\text{iload } n} (\mathsf{Int} \cdot S, R)$$

if $0 \leq n < M_{reg}$ and $R(n) = \mathsf{Int}$ and $|S| < M_{stack}$

$$(\mathsf{Int} \cdot S, R) \xrightarrow{\text{istore } n} (S, R\{n \mapsto \mathsf{Int}\})$$

if $0 \leq n < M_{reg}$

$$(\mathsf{Int} \cdot S, R) \xrightarrow{\mathsf{ifle}\ n} (S, R)$$

if n is a valid instruction location

$$(S, R) \xrightarrow{\mathsf{goto}\ n} (S, R)$$

if n is a valid instruction location

$$(S, R) \xrightarrow{\textsf{aconst\_null}} (\textsf{Null} \cdot S, R)$$

$$\text{if } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\mathsf{aconst\_null}} (\mathsf{Null} \cdot S, R)$$

$$\text{if } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\mathsf{aload}\ n} (R(n) \cdot S, R)$$

$$\text{if } 0 \leq n < M_{reg} \text{ and } R(n) \sqsubseteq \mathsf{Object} \text{ and } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\textsf{aconst\_null}} (\textsf{Null} \cdot S, R)$$

$$\text{if } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\textsf{aload}\ n} (R(n) \cdot S, R)$$

$$\text{if } 0 \leq n < M_{reg} \text{ and } R(n) \sqsubseteq \textsf{Object} \text{ and } |S| < M_{stack}$$

$$(\tau \cdot S, R) \xrightarrow{\textsf{astore}\ n} (S, R\{n \mapsto \tau\})$$

$$\text{if } 0 \leq n < M_{reg} \text{ and } \tau \sqsubseteq \textsf{Object}$$

Accessing fields and methods

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

$$\text{if } \tau' \sqsubseteq C$$

Accessing fields and methods

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

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

Accessing fields and methods

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

$$\text{if } \tau' \sqsubseteq C$$

$$(\tau_1 \cdot \tau_2 \cdot S, R) \xrightarrow{\textsf{putfield } C.f.\tau} (S, R)$$

$$\text{if } \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq C$$

$$(\tau'_n \cdot \ldots \cdot \tau'_1 \cdot S, R) \xrightarrow{\textsf{invokestatic } C.m.\sigma} (\tau \cdot S, R)$$

$$\text{if } \sigma = \tau(\tau_1, \ldots, \tau_n), \tau'_i \sqsubseteq \tau_i \text{ for } 1 \leq i \leq n \text{ and } |\tau \cdot S| \leq M_{stack}$$

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 $\bot \sqcup (R, S) = (R, S)$.

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 $\pi : pc_1 \to pc_2$ 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 $\pi : (S_0, R_0) \to (S_k, R_k)$.

For every valid location pc we define

Merge Over All Paths (MOP):

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

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:

$$r ::= \mathsf{r1} \mid \ldots \mid \mathsf{rk}$$

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:

$$r ::= \mathsf{r1} \mid \ldots \mid \mathsf{rk}$$

Operands:

$$\begin{array}{rll} \nu ::= & & \\ & n & \text{integer} \\ & \mid l & \text{label} \\ & \mid r & \text{register} \end{array}$$

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

Instructions

$$\iota ::=$$

$r_d := \nu$	assignment
$\mid r_d := r_s + \nu$	addition
$\mid \textsf{if}\ r\ \textsf{jump}\ \nu$	conditional jump

Instructions

$$\iota ::=$$

$$\begin{array}{ll} r_d := \nu & \text{assignment} \\ \mid r_d := r_s + \nu & \text{addition} \\ \mid \textsf{if } r \textsf{ jump } \nu & \text{conditional jump} \end{array}$$

instruction sequences

$$I ::= \textsf{ jump } \nu \mid \iota ; I$$

Instructions

$$\iota ::=$$

$$\begin{array}{ll} r_d := \nu & \text{assignment} \\ \mid r_d := r_s + \nu & \text{addition} \\ \mid \mathsf{if}\ r\ \mathsf{jump}\ \nu & \text{conditional jump} \end{array}$$

instruction sequences

$$I ::= \mathsf{jump}\ \nu \mid \iota; I$$

- Instruction sequences have at the end an unconditional jump to another instruction sequence pointer to by some label, and other instructions before.
- As yet, no infinite memory (except for code).

An example for computing square: r4 contains the return address

$$
\begin{array}{ll}
\mathsf{square}: & \mathsf{r3} := \mathsf{0}; \\
 & \mathsf{r2} := \mathsf{r1}; \\
 & \mathsf{jump\ loop} \\
\mathsf{loop}: & \mathsf{if\ r1\ jump\ done}; \\
 & \mathsf{r3} := \mathsf{r2} + \mathsf{r3}; \\
 & \mathsf{r1} := \mathsf{r1} + -\mathsf{1}; \\
 & \mathsf{jump\ loop} \\
\mathsf{done}: & \mathsf{jump\ r4}
\end{array}
$$

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