Types in Programming Languages (7)

Christian Urban

http://www4.in.tum.de/lehre/vorlesungen/types/WS0607/
Previously

untrapped errors
e.g. access of an array outside its bounds; jumping to a legal address

trapped errors
e.g. division by zero; jumping to an illegal address

evil

annoying
Previously

untrapped errors

e.g. access of an array outside its bounds; jumping to a legal address

trapped errors

e.g. division by zero; jumping to an illegal address

A programming language is called safe if no untrapped errors can occur. Safety can be achieved by run-time checks or static checks.

Munich, 13. December 2006 - p.2 (2/5)
Previously

untrapped errors
e.g. access of an array outside its bounds; jumping to a legal address

trapped errors
e.g. division by zero; jumping to an illegal address

evil

Forbidden errors include all untrapped errors and some trapped ones. A strongly typed programming language prevents all forbidden errors.
Previously

untrapped errors

- e.g. access of an array outside its bounds;
- jumping to a legal address

trapped errors

- e.g. division by zero;
- jumping to an illegal address

A weakly typed programming language prevents some untrapped errors, but not all; C, C++ have features that make them weakly typed.
Previously

untrapped errors

- e.g. access of an array outside its bounds; jumping to a legal address

trapped errors

- e.g. division by zero; jumping to an illegal address

<table>
<thead>
<tr>
<th></th>
<th>Typed</th>
<th>Untyped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe</td>
<td>SML, Java</td>
<td>LISP</td>
</tr>
<tr>
<td>Unsafe</td>
<td>C, C++</td>
<td>Assembler</td>
</tr>
</tbody>
</table>
Real World-Compilers

So far we said that a program should type-check, and then we forget about types (not always possible because of dynamic checks)

This is however not what happens in practice:

- an optimising compiler for a high-level language might make as many as 20 passes over a single program
- many optimisations require type-information to succeed (direct register allocation for integer operations)
- a compiler often translates between many intermediate languages (type-information helps to stay sane)
Safety in the Target Lang.

The target language (e.g. Java bytecode or Microsoft's Common Language infrastructure) might be typed.

In Java bytecode the types of the parameters of all instructions are known and the verifier ensures they are correct.

This ensures there are no operand stack overflows or underflows; pointer arithmetic is not arbitrary.

Only when the bytecode is run, most checks are not needed anymore.

The ultimate goal is that you can run untrusted code on your machine.
We want to ensure the property of control-flow safety of “assembler programs”: A program cannot jump to an arbitrary address, but only to a well-defined subset of possible entry points.

Greg Morrisett calls this language TAL-0 (Typed Assembly Language) and describes it in the book on advanced topics on types and programming languages.
Language

**Registers**

\[ r ::= r_1 \mid \ldots \mid r_k \]

**Operands**

\[ v ::= n \quad \text{integer literal} \]
\[ \mid l \quad \text{label or pointer} \]
\[ \mid r \quad \text{register} \]

**Instructions**

\[ i ::= r ::= v \]
\[ \mid r ::= r + v \]
\[ \mid \text{if } r \ \text{jump } v \]

**Programs**

\[ p ::= \text{jump } v \]
\[ \mid i; p \]
Example

The calculation of the product of \( r_1 \) and \( r_2 \), placing the result in \( r_3 \); return to an address assumed to be in \( r_4 \):

\[
\begin{align*}
\text{prod:} & \quad r_3 := 0 \quad \% \quad res := 0 \\
 & \quad \text{jump loop}
\end{align*}
\]

\[
\begin{align*}
\text{loop:} & \quad \text{if } r_1 \text{ jump done} \quad \% \quad \text{if } a = 0 \text{ goto done} \\
 & \quad r_3 := r_2 + r_3 \quad \% \quad res := res + b \\
 & \quad r_1 := r_1 + (-1) \quad \% \quad a := a - 1 \\
 & \quad \text{jump loop}
\end{align*}
\]

\[
\begin{align*}
\text{done:} & \quad \text{jump } r_4 \quad \% \quad \text{return}
\end{align*}
\]
Machine States

Machine states are triples \((H, R, p)\)

Heaps

\[
\{l_1 := p_1, \ldots, l_m := p_m\}
\]
Heaps

\[ H = \{ \text{prod} = p_{\text{prod}}, \text{loop} = p_{\text{loop}}, \text{done} = p_{\text{done}} \} \]
Machine States

- Machine states are triples $(H, R, p)$

- Heaps

$$\{l_1 := p_1, \ldots, l_m := p_m\}$$

- Register files

$$\{r_1 := v_1, \ldots, r_n := v_n\}$$

- Safety property is that no machine state is stuck (for example jump 42 is stuck).
Transitions

- **Jump**

\[ H(v) = p \]

\[(H, R, \text{jump } v) \rightarrow (H, R, p)\]

- **Mov**

\[(H, R, r := v; p) \rightarrow (H, R[r := v], p)\]

- **Add**

\[ R(r') = n \quad R(v) = n' \]

\[(H, R, r := r' + v; p) \rightarrow (H, R[r := r' + v], p)\]

- **If-eq**

\[ R(r) = 0 \quad H(v) = p' \]

\[(H, R, \text{if } r \text{ jump } v; p) \rightarrow (H, R, p')\]

- **If-neq**

\[ R(r) \neq 0 \]

\[(H, R, \text{if } r \text{ jump } v; p) \rightarrow (H, R, p)\]
Type System

- Any well-typed “machine” cannot get stuck (remember jump 42 should not be a well-typed program).

- Types

  \[ T ::= \begin{array}{c}
  \text{int} \\
  X \\
  \forall X.T \\
  \text{code}(\Gamma)
  \end{array} \]

- \[ \Gamma ::= \{ r_1 : T_1, \ldots, r_n : T_n \} : \text{these are register file types (in a minute)} \]
Example

prod: \[ r_3 := 0 \]
    jump loop

loop: if \( r_1 \) jump done
    \( r_3 := r_2 + r_3 \)
    \( r_1 := r_1 + (-1) \)
    jump loop

done: jump \( r_4 \)

\( \Gamma \) contains the “assumptions” we make about the code
\[ \{ r_1, r_2, r_3 : \text{int}, r_4 : \forall X.\text{code}\{r_1, r_2, r_3 : \text{int}, r_4 : X\}\} \]

They will be recorded in \( \Delta \), for example
\[ \{ \text{prod : code(\( \Gamma \)), loop : code(\( \Gamma \)), done : code(\( \Gamma \))} \} \]
Judgements (I)

We will have several kinds of judgments:

- Integer literal

\[ \Delta \vdash n : \text{int} \]

- Label

\[ (l : T) \in \Delta \quad \frac{}{\Delta \vdash l : T} \]

(We want to have unique labels.)
Judgements (II)

**Register**

\[
(r : T) \in \Gamma \\
\Delta; \Gamma \vdash r : T
\]

**Value (non-register)**

\[
\Delta \vdash v : T \\
\Delta; \Gamma \vdash v : T
\]

**Type-instantiation**

\[
\Delta; \Gamma \vdash v : \forall X.T \\
\Delta; \Gamma \vdash v : T'[X := T']
\]
Instructions will be dealt with by

\[ \Delta \vdash i : \Gamma_{\text{in}} \rightarrow \Gamma_{\text{out}} \]

**Mov**

\[ \Delta; \Gamma \vdash v : T \]

\[ \Delta \vdash r := v : \Gamma \rightarrow \Gamma[r : T] \]

**Add**

\[ \Delta; \Gamma \vdash r' : \text{int} \quad \Delta; \Gamma \vdash v : \text{int} \]

\[ \Delta \vdash r := r' + v : \Gamma \rightarrow \Gamma[r : \text{int}] \]

**If**

\[ \Delta; \Gamma \vdash r : \text{int} \quad \Delta; \Gamma \vdash v : \text{code}(\Gamma) \]

\[ \Delta \vdash \text{if } r \text{ jump } v : \Gamma \rightarrow \Gamma \]
Judgements (IV)

- **programs** (instruction sequences)
  \[ \Delta \vdash p : \text{code}(\Gamma) \]

- **Jump**
  \[ \Delta; \Gamma \vdash \nu : \text{code}(\Gamma) \]
  \[ \Delta \vdash \text{jump } \nu : \text{code}(\Gamma) \]

- **Seq**
  \[ \Gamma \vdash i : \Gamma \rightarrow \Gamma' \]
  \[ \Delta \vdash p : \text{code}(\Gamma') \]
  \[ \Delta; \Gamma \vdash i; p : \text{code}(\Gamma) \]
Examples

Let $\Gamma$ be
\[
\{ r_1, r_2, r_3 : \text{int}, r_4 : \forall X.\text{code}\{ r_1, r_2, r_3 : \text{int}, r_4 : X \} \}\]

Let $\Delta$ be
\[
\{ \text{prod} : \text{code}(\Gamma), \text{loop} : \text{code}(\Gamma), \text{done} : \text{code}(\Gamma) \}\]

Derivable judgements:

- $\Delta \vdash \text{if } r_1 \text{ jump done} : \Gamma \rightarrow \Gamma$
- $\Delta \vdash r_3 := r_2 + r_3 : \Gamma \rightarrow \Gamma$
- $\Delta \vdash r_1 := r_1 + (-1) : \Gamma \rightarrow \Gamma$
- $\Delta \vdash \text{jump loop} : \text{code}(\Gamma)$

So we showed $\Delta \vdash p_{\text{loop}} : \text{code}(\Gamma)$
A register file is well-typed, written \( \Delta \vdash R : \Gamma \), if for all \((r : T)\) in \(R\)

\[
\Delta; \Gamma \vdash r : T
\]

A heap is well-typed, written \( \vdash H : \Delta \), if for all \(l : T\) in \(\Delta\)

\[
\Delta \vdash H(l) : T
\]

and the \(T\) does not contain any free type-variables.
Well-Typedness

The types avoid to jump to an integer or an undefined label — however the situation is more complicated than is solvable by tags.

We can have

\[
\begin{align*}
\text{foo:} & \quad r_1 := \text{bar} \\
\text{jump} & \quad r_1 \\
\text{bar:} & \quad \ldots
\end{align*}
\]
Polymorphism even allows us

\{ r_1 : \text{int}, \ldots \} \quad \text{jump bar}

\{ r_1 : \text{code}(\ldots), \ldots \} \quad \text{jump bar}

where the type of bar is

\forall X. \text{code}(r_1 : X, \ldots).
We can show that given a well-typed machine state \( M \) then \( M \) cannot get stuck (i.e. jump to an integer or an undefined label).

Proof-Outline: \( M \) is not immediately stuck and if \( M \rightarrow M' \) then \( M' \) is also well-typed.

Question: given a machine state \( M = (H, R, p) \) can one find a \( \Delta \) and \( \Gamma \) such that \( \Delta \vdash p : \text{code}(\Gamma) \) etc?

Answer: We do not know. (Likely not.)

The compiler has to give enough information during the compilation process so that the bytecode only needs to be “type-verified” — type-inference is too hard.
More Next Week

- Slides at the end of

  http://www4.in.tum.de/lehre/vorlesungen/types/WS0607/

  There is also an appraisal form where you can complain anonymously.

- You can say whether the lecture was too easy, too quiet, too hard to follow, too chaotic and so on. You can also comment on things I should repeat.