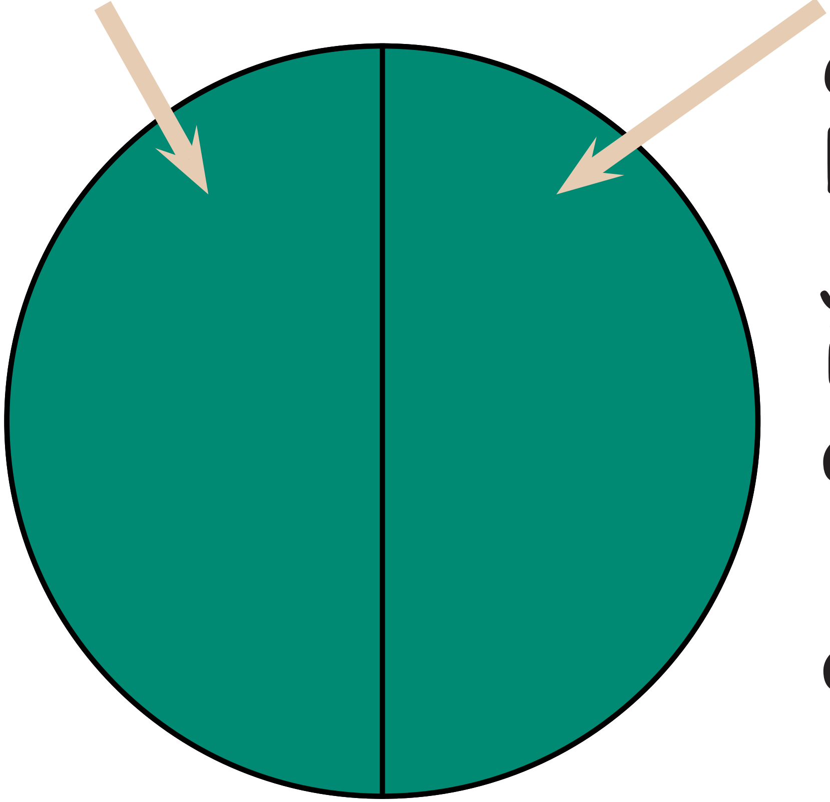
# Types
## in Programming Languages (7)

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

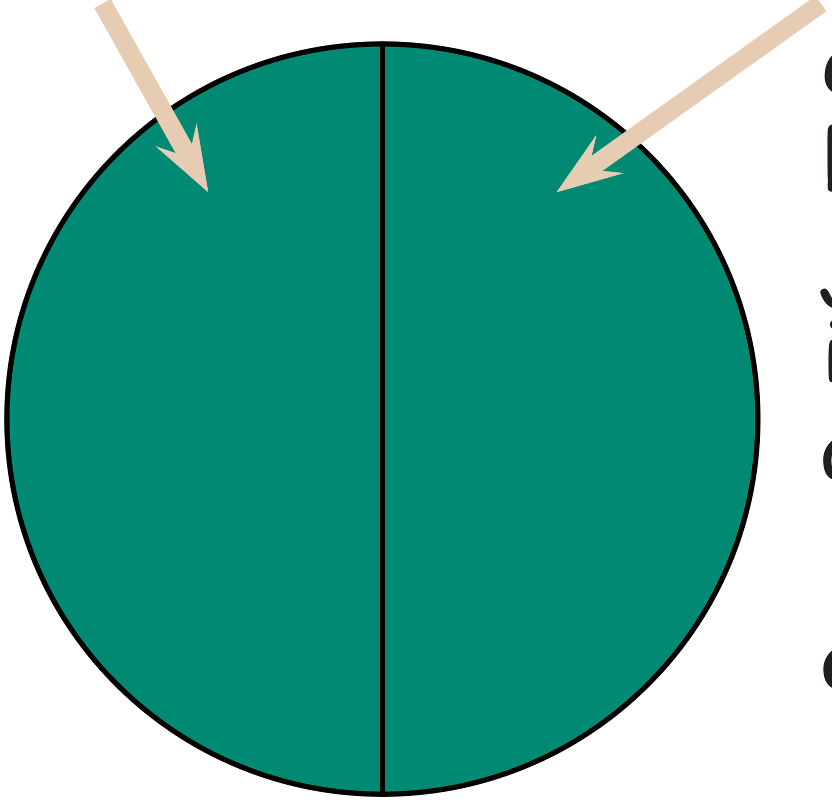
**evil**

**trapped errors**

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

**annoying**

# Previously



**untrapped errors**

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

**evil**

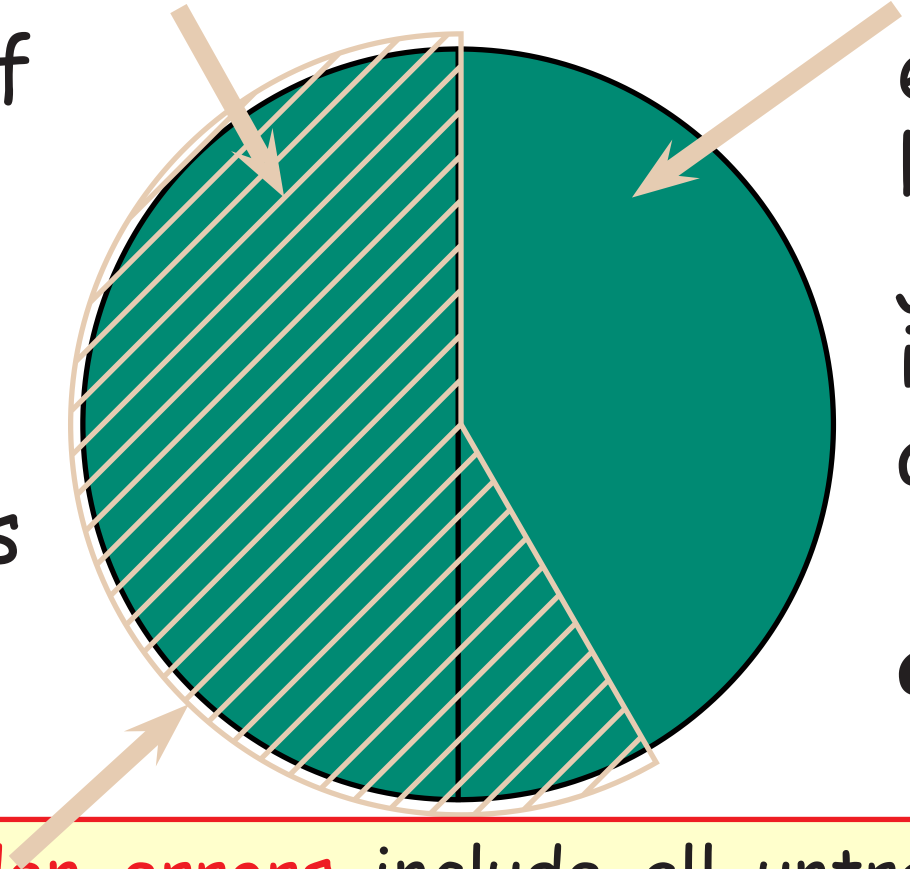
**trapped errors**

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

**annoying**

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

# Previously



**untrapped errors**

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

**evil**

**trapped errors**

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

**annoying**

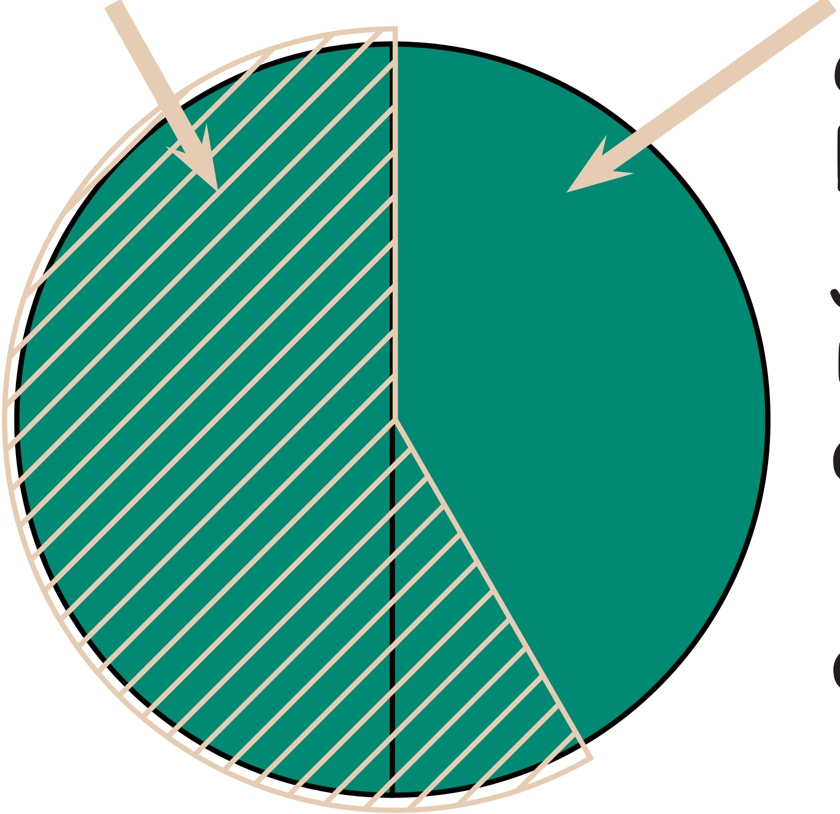
Forbidden errors include all untrapped errors and some trapped ones. A strongly typed programming language prevents all forbidden errors.

# Previously

**untrapped errors**

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

**evil**



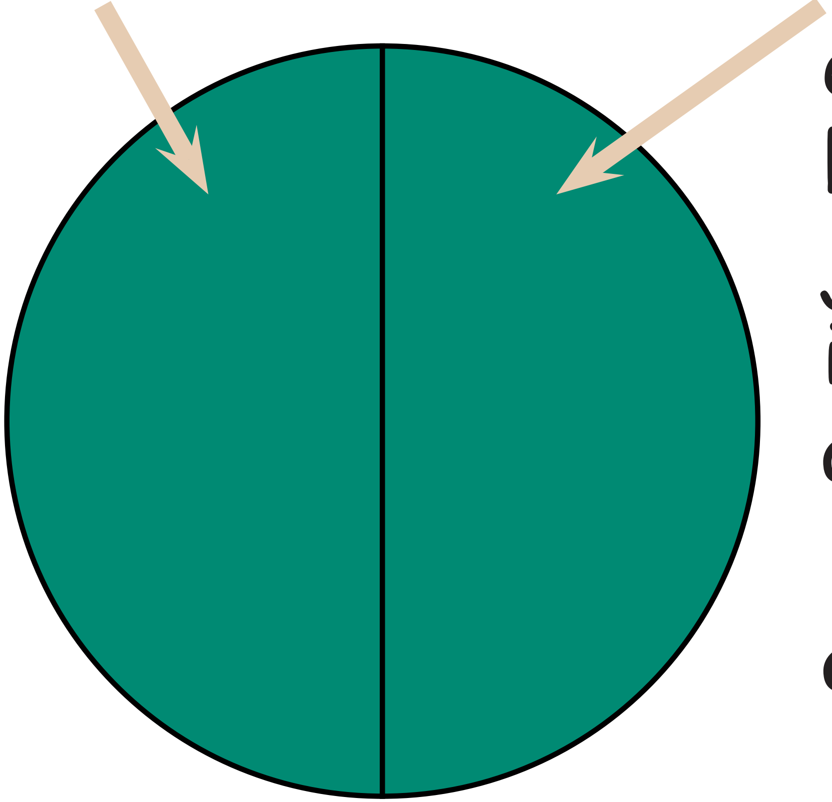
**trapped errors**

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

**annoying**

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

**evil**

trapped errors

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

**annoying**

| | Typed | Untyped |
|---|---|---|
| Safe | SML, Java | LISP |
| Unsafe | C, C++ | Assembler |

# 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.

# Example We Shall Look At

- 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

$$\begin{array}{lcll} v & ::= & n & \text{integer literal} \\ & \mid & l & \text{label or pointer} \\ & \mid & r & \text{register} \end{array}$$

- Instructions

$$\begin{array}{lcl} i & ::= & r := v \\ & \mid & r := r + v \\ & \mid & \text{if } r \text{ jump } v \end{array}$$

- Programs

$$\begin{array}{lcl} p & ::= & \text{jump } v \\ & \mid & i; p \end{array}$$

# 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 $r_4$:

```
prod:   $r_3 := 0$                      % $res := 0$
        jump loop
loop:   if $r_1$ jump done              % if $a = 0$ goto done
        $r_3 := r_2 + r_3$              % $res := res + b$
        $r_1 := r_1 + (-1)$             % $a := a - 1$
        jump loop
done:   jump $r_4$                      % return
```

# Machine States

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

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

# Heaps

$$p_{\mathsf{prod}} \left\{ \begin{array}{ll} \mathsf{prod:} & r_3 := 0 \\ & \mathsf{jump\ loop} \end{array} \right.$$

$$p_{\mathsf{loop}} \left\{ \begin{array}{ll} \mathsf{loop:} & \mathsf{if}\ r_1\ \mathsf{jump\ done} \\ & r_3 := r_2 + r_3 \\ & r_1 := r_1 + (-1) \\ & \mathsf{jump\ loop} \end{array} \right.$$

$$p_{\mathsf{done}} \left\{ \begin{array}{ll} \mathsf{done:} & \mathsf{jump}\ r_4 \end{array} \right.$$

$$H = \{\mathsf{prod} = p_{\mathsf{prod}}, \mathsf{loop} = p_{\mathsf{loop}}, \mathsf{done} = p_{\mathsf{done}}\}$$

# 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

$$\frac{H(v) = p}{(H, R, \textsf{jump}\ v) \to (H, R, p)}$$

- Mov

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

- Add

$$\frac{R(r') = n \quad R(v) = n'}{(H, R, r := r' + v; p) \to (H, R[r := r' + v], p)}$$

- If-eq

$$\frac{R(r) = 0 \quad H(v) = p'}{(H, R, \textsf{if}\ r\ \textsf{jump}\ v; p) \to (H, R, p')}$$

- If-neq

$$\frac{R(r) \neq 0}{(H, R, \textsf{if}\ r\ \textsf{jump}\ v; p) \to (H, R, p)}$$

# Type System

- Any well-typed "machine" cannot get stuck (remember jump $42$ should not be a well-typed program).
- Types

$$
\begin{array}{rcll}
T & ::= & \mathsf{int} & \\
 & | & X & \text{type-variables} \\
 & | & \forall X.T & \text{polymorphic types} \\
 & | & \mathsf{code}(\Gamma) & \text{code labels}
\end{array}
$$

- $\Gamma ::= \{r_1 : T_1, \ldots, r_n : T_n\}$: 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 $r_4$
```

- $\mathit{\Gamma}$ contains the "assumptions" we make about the code

$$\{r_1, r_2, r_3 : \mathsf{int}, r_4 : \forall X.\mathsf{code}\{r_1, r_2, r_3 : \mathsf{int}, r_4 : X\}\}$$

- They will be recorded in $\Delta$, for example

$$\{\mathsf{prod} : \mathsf{code}(\mathit{\Gamma}), \mathsf{loop} : \mathsf{code}(\mathit{\Gamma}), \mathsf{done} : \mathsf{code}(\mathit{\Gamma})\}$$

# Judgements (I)

- We will have **several** kinds of judgments:
- Integer literal

$$\Delta \vdash n : \mathsf{int}$$

- Label

$$\frac{(l : T) \in \Delta}{\Delta \vdash l : T}$$

(We want to have unique labels.)

# Judgements (II)

- Register

$$\frac{(r : T) \in \Gamma}{\Delta; \Gamma \vdash r : T}$$

- Value (non-register)

$$\frac{\Delta \vdash v : T}{\Delta; \Gamma \vdash v : T}$$

- Type-instantiation

$$\frac{\Delta; \Gamma \vdash v : \forall X.T}{\Delta; \Gamma \vdash v : T[X := T']}$$

# Judgements (III)

- Instructions will be dealt with by

$$\Delta \vdash i : \Gamma_{\mathsf{in}} \rightarrow \Gamma_{\mathsf{out}}$$

- Mov

$$\frac{\Delta; \Gamma \vdash v : T}{\Delta \vdash r := v : \Gamma \rightarrow \Gamma[r : T]}$$

- Add

$$\frac{\Delta; \Gamma \vdash r' : \mathsf{int} \quad \Delta; \Gamma \vdash v : \mathsf{int}}{\Delta \vdash r := r' + v : \Gamma \rightarrow \Gamma[r : \mathsf{int}]}$$

- If

$$\frac{\Delta; \Gamma \vdash r : \mathsf{int} \quad \Delta; \Gamma \vdash v : \mathsf{code}(\Gamma)}{\Delta \vdash \mathsf{if}\ r\ \mathsf{jump}\ v : \Gamma \rightarrow \Gamma}$$

# Judgements (IV)

- programs (instruction sequences)

$$\Delta \vdash p : \mathsf{code}(\Gamma)$$

- Jump

$$\frac{\Delta; \Gamma \vdash v : \mathsf{code}(\Gamma)}{\Delta \vdash \mathsf{jump}\ v : \mathsf{code}(\Gamma)}$$

- Seq

$$\frac{\Gamma \vdash i : \Gamma \to \Gamma' \quad \Delta \vdash p : \mathsf{code}(\Gamma')}{\Delta; \Gamma \vdash i; p : \mathsf{code}(\Gamma)}$$

# Examples

- Let $\Gamma$ be

$$\{r_1, r_2, r_3 : \mathsf{int}, r_4 : \forall X.\mathsf{code}\{r_1, r_2, r_3 : \mathsf{int}, r_4 : X\}\}$$

- Let $\Delta$ be

$$\{\mathsf{prod} : \mathsf{code}(\Gamma), \mathsf{loop} : \mathsf{code}(\Gamma), \mathsf{done} : \mathsf{code}(\Gamma)\}$$

- Derivable judgements:
  - $\Delta \vdash \mathsf{if}\ r_1\ \mathsf{jump\ done} : \Gamma \to \Gamma$
  - $\Delta \vdash r_3 := r_2 + r_3 : \Gamma \to \Gamma$
  - $\Delta \vdash r_1 := r_1 + (-1) : \Gamma \to \Gamma$
  - $\Delta \vdash \mathsf{jump\ loop} : \mathsf{code}(\Gamma)$
- So we showed $\Delta \vdash p_{\mathsf{loop}} : \mathsf{code}(\Gamma)$

# Loose Ends

- 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

foo: $r_1 := \text{bar}$

jump $r_1$

bar: $\ldots$

# Well-Typedness

- Polymorphism even allows us

  $$\{r_1 : \mathsf{int}, \ldots\}$$
  jump bar

  $$\{r_1 : \mathsf{code}(\ldots), \ldots\}$$
  jump bar

  where the type of bar is

  $$\forall X.\mathsf{code}(r_1 : X, \ldots).$$

# Next Time

- 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 $\Delta \vdash p : \mathsf{code}(\Gamma)$ 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.