DYNAMIC TYPING versus STATIC TYPING

Instead of checking the type information at compile time, we could

- check all errors at run-time
- treat type errors like division by zero etc

Advantages:

- more flexibility: e.g., expressions like
  
  \[(\text{if True then 5 else False}) + 1\]
  
  are no problem.
- heterogeneous data structures are possible:
  
  \[\{1.0, 3, [2], \text{True}, \text{False}\}\]

Disadvantages:

- programming errors are detected later in the development cycle — during testing (or not) instead of during compilation
- programmer has to keep track of data type of each object
- overhead of run-time checks

DYNAMICALY TYPED MINHS

Changes:

- **Language:**
  - throw out type annotations for functions and variables

- **Static Semantics:**
  - throw out type checking
  - check only if expressions are well-formed

- **Dynamic Semantics:**
  - add run-time checks for function application, application of built-in functions, and conditions of if-expressions

Function application:

\[e_1 \mapsto e'_1\]
\[\text{apply}(e_1, e_2) \mapsto \text{apply}(e'_1, e_2)\]
\[e_2 \mapsto e'_2\]
\[\text{apply}(\text{letfun}(\ldots), e_2) \mapsto \text{apply}(\text{letfun}(\ldots), e'_2)\]
\[\text{apply}(\text{letfun}(f.x.e), v) \mapsto (\text{letfun}(f.x.e)\{v/x\})[v]/e_1\]
Runtime error checks:

\[
\begin{align*}
& v \neq \text{letfun(...)} \\
\Rightarrow & \quad \text{apply}(v, v') \rightarrow \text{error}
\end{align*}
\]

\[
\begin{align*}
& \text{apply}(\text{error}, v') \rightarrow \text{error} \\
& \text{apply}(v, \text{error}) \rightarrow \text{error}
\end{align*}
\]

Progress and preservation still holds!

- **Preservation:** The only static property we check for, well-formedness, is preserved under reduction.
- **Progress:** No stuck states, as all type errors are handled by dynamic checks and result in error-value

**Implementing Dynamic Typing**

How can we implement runtime type checking rules?
- **Obvious representations do not lend themselves well to such comparisons.**
  - not possible to decide whether a 32-bit field on a certain architecture represents an integer value or a pointer to something
- **We need a value representation which allows us to easily determine type of value.**
  - values are “tagged”, or labeled with their type
- **It’s actually not run-time type checking, but run-time tag-checking!**

We can make the tagging explicit in the language: we distinguish between

- **tagged values:** all values in the program are tagged values, expressions other than values are never tagged
- **untagged values:** arise only during evaluation, not as expressions in a program

\[
\begin{align*}
\text{tagged values} & ::= \text{Int}(n) \mid \text{Bool(True)} \mid \text{Bool(False)} \mid \text{Fun(letfun}(f.x.e)) \\
\text{untagged values} & ::= n \mid \text{True} \mid \text{False} \mid \text{letfun}(f.x.e)
\end{align*}
\]
**TAG-CHECKING RULES**

Check if value has certain type, retrieve content:

- `Fun(u)` \( \text{isFun} \ u \)
- `Bool(u)` \( \text{isNotFun} \)
- `Int(u)` \( \text{isNotFun} \)
- ...  
- `Int(u)` \( \text{isInt} \ u \)
- ... ...

Similar rules for booleans and integers.

**Application:**

\[
\begin{align*}
\text{v isFun} & \quad \text{letfun}(f, x, e) \\
\text{apply}(v, v') & \rightarrow \{v/f\}(v'/x)e
\end{align*}
\]

\[
\begin{align*}
\text{v isNotFun} & \\
\text{apply}(v, v') & \rightarrow \text{error}
\end{align*}
\]

**Primitive operations:**

\[
\begin{align*}
v_1 \text{ isInt } n_1 & \quad v_2 \text{ isInt } n_2 \\
\text{plus}(v_1, v_2) & \rightarrow \text{Int}(n_1 + n_2)
\end{align*}
\]

**Observations:**

\(
\begin{align*}
\rightarrow \quad & \text{Dynamic typing incurs high costs (run-time and memory)} \\
\rightarrow & \text{Overhead is independent of whether dynamic typing is actually required or not} \\
\rightarrow & \text{Usually, dynamic typing is only necessary for a small portion of a program}
\end{align*}
\)

**DYNAMIC TYPING AS STATIC TYPING**

We can define our tagged data type in a statically typed language:

\[
\text{data Tagged} = \text{IntT} \text{ Int} \\
| \text{BoolT} \text{ Bool} \\
| \text{FunT} (\text{Tagged} \rightarrow \text{Tagged})
\]

checked_add:: Tagged -> Tagged -> Tagged
checked_add v1 v2 =
  case (v1, v2) of
    (IntT n1, IntT n2) -> IntT (n1 + n2)
    _                  -> error "Type error"

Slide 13
checked_apply:: Tagged -> Tagged -> Tagged
checked_apply f x =
  case f of
    FunT f' -> f x
    _       -> error "Type error"