**DYNAMIC TYPING versus STATIC TYPING**

Instead of checking the type information at compile time, we could
Dynamic Typing versus Static Typing

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

→ check all errors at run-time
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

   (if True then 5 else False) + 1

   are no problem.

→ heterogeneous data structures are possible:

   [1.0, 3, [2], True, 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
DYNAMICALLY TYPED MINHS

Changes:

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

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

\[ \frac{v \neq \text{letfun}(\ldots)}{\text{apply}(v, v') \mapsto \text{error}} \]

\[ \text{apply}(\text{error}, v') \mapsto \text{error} \quad \text{apply}(v, \text{error}) \mapsto \text{error} \]
Progress and preservation still holds!

➡️ **Preservation:**

➡️ **Progress:**
Progress and preservation still holds!

➔ **Preservation:** The only static property we check for, well-formedness, is preserved under reduction.

➔ **Progress:**
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. It’s actually not run-time checking, but run-time tag-checking!
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
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
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
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

\[
\begin{align*}
tagged \; values \ &:= \int(n) \mid \bool(True) \mid \bool(False) \mid \fun(letfun(f.x.e)) \\
untagged \; values \ &:= \ n \mid \ True \mid \ False \mid \ letfun(f.x.e)
\end{align*}
\]
TAG-CHECKING Rules

Check if value has certain type, retrieve content:
TAG-CHECKING RULES

Check if value has certain type, retrieve content:

- Fun($u$) **isFun** $u$
- Bool($u$) **isNotFun**
- Int($u$) **isNotFun**
- ...  
- Int($u$) **isInt** $u$
- ...
**Tag-checking Rules**

Check if value has certain type, retrieve content:

- `Fun(u)` *isFun u*
- `Bool(u)` *isNotFun*
- `Int(u)` *isNotFun*
- …
- `Int(u)` *isInt u*
- …

Similar rules for booleans and integers.
Application:

\[ v \text{ isFun } \text{letfun}(f.x.e) \]

\[ \text{apply}(v, v') \mapsto \{v/f\}\{v'/x\}e \]

\[ v \text{ isNotFun} \]

\[ \text{apply}(v, v') \mapsto \text{error} \]
Application:

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

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

 Primitive operations:

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

- Dynamic typing incurs high costs (run-time and memory)
- Overhead is independent of whether dynamic typing is actually required or not
- Usually, dynamic typing is only necessary for a small portion of a program
DYNAMIC TYPING AS STATIC TYPING

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

```haskell
data Tagged = IntT Int
            | BoolT Bool
            | FunT (Tagged -> Tagged)
```
checked_add :: Tagged -> Tagged -> Tagged
checked_add v1 v2 =
    case (v1, v2) of
        (IntT n1, IntT n2) -> IntT (n1 + n2)
        _                     -> error "Type error"

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