Motivation for typed languages

• Untyped Languages: perform *any* operation on *any* data.

• Example: Assembly
  
  movi 5 r0 // Move integer 5 (some representation) to r0
  addf 3.6 r0 // Treat bit representation in r0 as a
               // Floating point representation and add 3.6
               // to it.

  Result? You can be sure that r0 does not contain 8.6!

• (+) Flexibility : “I can do anything I want to”
• (−) Ease of Error Checking. (programs are prone to errors, especially huge ones).
Motivation for typed languages

Typed Languages:

• A type represents a set of values. Programs / procedures / operators are functions from an input type to an output type.

• Type Checking is the activity of ensuring that the operands / arguments of an operator / procedure are of compatible type through the use of a set of rules for associating a type with every expression in the language. (These rules are known as the type system).

• A type error results when an operator is applied to an operand of inappropriate/incompatible type.

• Output of a type system:
  – There are type-errors (wrt type system) => Program is NOT type-safe.
  – There are no type-errors (wrt type system) => Program is type-safe.
How to type-check?

Definition:
- **Type statements** are of the form:
  \[
  \text{<expr>} : \text{<type>}
  \]
  meaning that an expression \text{<expr>} ‘is-of-the-type’ (the ‘:’ symbol) \text{<type>}.
- Examples:
  - 3 : int
  - 3+4 : int
  - 3.14 : real
  - “abc” : String
  - while (x < 5) {x++;} : Stmt
How to type-check?

Definition:

- **Type rules** are of the form:

  \[
  \frac{e_1 : t_1 \quad e_2 : t_2 \quad \ldots \quad e_n : t_n}{f \ e_1 \ e_2 \ \ldots \ e_n : t} \quad \text{(rule name)}
  \]

  where each \( e_i : t_i \) is a type statement, \( n \geq 0 \).

  The rule is interpreted as “**IF** \( e_1 \) is of type \( t_1 \) and \( \ldots \) and \( e_n \) is of type \( t_n \) **THEN** \( f \ e_1 \ e_2 \ \ldots \ e_n \) is of type \( t \).”
How to type-check?

Examples of type rules:

• Rule for constants:

\[
\begin{array}{llll}
1 : \text{int} & 2 : \text{int} & 3 : \text{int} & \cdots \\
\end{array}
\]

• Rule for addition:

\[
\begin{array}{ll}
E_1 : \text{int} & \quad E_2 : \text{int} \\
\hline
E_1 + E_2 : \text{int} \\
\end{array}
\]

• Rule for boolean comparison:

\[
\begin{array}{ll}
E_1 : \text{int} & \quad E_2 : \text{int} \\
\hline
E_1 == E_2 : \text{bool} \\
\end{array}
\]
How to type-check?

- Rule for assignment statement:

Examples of type rules:

\[
\begin{align*}
&x : T \\
&E : T \\
\hline \\
&x := E; : Stmt
\end{align*}
\]

- Rule for if-statement:

\[
\begin{align*}
&E_1 : \text{Bool} \\
&S_1 : \text{Stmt} \\
&S_2 : \text{Stmt} \\
\hline \\
&\text{if } (E_1) \{S_1\} \text{ else } \{S_2\} : \text{Stmt}
\end{align*}
\]
How to type-check?

• Rules of Type Checking
  – Type of value => known in advance
  – Type of variable => known in the declaration
  – Type of function => known from the type of arguments (in declaration) and type of result (also in declaration).
  – Type of expression => inferred from sub-expression.
How to type-check?

• Given the program:

```
int x;
...
x := x+1;
...
```

...And Given the rules:

```
1 : int
2 : int
3 : int

E₁ : int    E₂ : int
--- (+) ---
E₁ + E₂ : int

E₁ : int    E₂ : int
--- (==) ---
E₁ == E₂ : bool

x : T    E : T
--- (:=) ---
E : Stmt
x := E;

E₁ : Bool
S₁ : Stmt
S₂ : Stmt
--- (if) ---
if (E₁) {S₁} else {S₂} : Stmt

x : int    1 : int
--- (+) ---
x := x+1 : Stmt
```

A program/expression is type-safe if we can construct a derivation tree to give a type for that program/expression.
How to type-check?

• Given the program:
  ```java
  int x;   float y;
  ...
  if (x == 3) {
    y := x;
  } else {
    x := x+1;
  }
  ...
  ```

  A program/expression is type-safe if we can construct a derivation tree to give a type for that program/expression.

  Follow the rules! Try to build tree. Cannot build tree => Not type safe

  ...And Given the rules:

  | 1: int | 2: int | 3: int | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E₁ : int</td>
<td>E₂ : int</td>
<td>(+)</td>
</tr>
<tr>
<td>E₁ + E₂ : int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₁ : int</td>
<td>E₂ : int</td>
<td>(==)</td>
</tr>
<tr>
<td>E₁ == E₂ : bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x : T</td>
<td>E : T</td>
<td>(:=)</td>
</tr>
<tr>
<td>x := E ; : Stmt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₁ : Bool</td>
<td>S₁ : Stmt</td>
<td>S₂ : Stmt</td>
</tr>
<tr>
<td>if (E₁) {S₁} else {S₂} : Stmt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  if (x==3) {y:=x;} else {x:=x+1;} : Stmt

  x==3 : Bool
  (==)
  y:=x; : Stmt

  x : int
  (==)
  x+1 : int
  (:=)
  if (x==3) {y:=x;} else {x:=x+1;} : Stmt

  x : int
  (==)
  1 : int
  (+)
  if (x==3) {y:=x;} else {x:=x+1;} : Stmt
How to type-check?

- **Dynamic** type checking
- **Static** type checking

- **Strongly typed**: we can detect all type errors statically (all \( f: S \rightarrow R \) *type safe*).
- **ML**: type declarations are not necessary if the interpretation is *unambiguous*. 
How to cater for Polymorphism

• Polymorphism = poly (many) + morph (form)

• **Polymorphism** is the ability of a data object to take on or assume many different forms (Polymorphic operations)
Polymorphism – Coercion

COERCION
A coercion is a operation that converts the type of an expression to another type. It is done automatically by the language compiler.
(If the programmer manually forces a type conversion, it’s called casting)

\[
\begin{array}{c}
E : \text{int} \\
\hline
\rightarrow F \text{ (Int-Float Coercion)} \\
E : \text{float}
\end{array}
\]

```c
int x; float y;
...
y := x;
...
```
Polymorphism – Coercion

Example of the use of COERCION

```plaintext
int x;    float y;
...
if (x == 3) {
    y := x;
} else {
    x := x+1;
}
...
```

Add in new rule...

```
E : int
E : float
(Int-Float Coercion)
```

```
x : int
3 : int
```

```
y : float
x : float
(Coercion)
```

```
x : int
1 : int
```

```
x == 3 : Bool
y := x : Stmt
```

```
if (x==3) {y:=x;} else {x:=x+1;} : Stmt
```

```
x : int
+1 : int
```

```
x := x+1; : Stmt
```

```
if (x==3) {y:=x;} else {x:=x+1;} : Stmt
```

```
E1 : int
E2 : int
```

```
E1 + E2 : int
(+)
```

```
E1 : int
E2 : int
```

```
E1 == E2 : bool
(=)
```

```
x : T
E : T
```

```
x := E;  : Stmt
(:=)
```

```
E1 : Bool
S1 : Stmt
S2 : Stmt
```

```
if (E1) {S1} else {S2} : Stmt
(if)
```

```
x : int
3 : int
```

```
y : float
x : float
```

```
x : int
x + 1 : int
```

```
x := x+1; : Stmt
```
Polymorphism – Coercion

Coercion

Widening

Widening coercion converts a value to a type that can include all of the values of the original type.

Widening is safe most of the time.

Narrowing

Narrowing coercion converts a value to a type that cannot store (even approximations of) all of the values of the original type.

Narrowing is unsafe. Information is lost during conversion of type.

Theoretically speaking, int ⊆ float
Polymorphism – Coercion

**Coercions**

(+) *Increase flexibility* in programming

• Example:

```java
float x, y, z;
int a, b, c;
```

• If I have no coercions, and I intend to add `y` and `a` and store in `x`, then writing…

```java
x = y + ((float) a);
```

...is too much of a hassle.

• Therefore coercion is good.
Polymorphism – Coercion

Coercions

(−) Decrease Reliability (error detection)

• Example:

```c
float x, y, z;
int a, b, c;
```

• If I have coercions and I intend to add \( x \) and \( y \) and store in \( z \), but I accidentally write…

```c
z = x + a;
```

…then my error will go undetected because the compiler will simply coerce the \( a \) to a float.

• Therefore coercion is bad.
Polymorphism – Coercion

Coercions:
• A lot of them: PL/I, Fortran, C, C++
• Fewer: Java (permits only widening)
• Very Few: Ada
Polymorphism – Overloading

OVERLOADING
An overloaded operation has different meanings, and different types, in different contexts.

\[ E_1 : \text{int} \quad E_2 : \text{int} \]
\[ \frac{E_1 + E_2 : \text{int}}{(+\text{-int})} \]

\[ E_1 : \text{float} \quad E_2 : \text{float} \]
\[ \frac{E_1 + E_2 : \text{float}}{(+\text{-float})} \]
Polymorphism – Overloading

Example of the use of Overloading

int x, y, z; float a, b, c;

... if (x == 3) {
  x := y + z;
} else {
  a := b + c;
}

... Add in new rule...

E₁ : float  E₂ : float  (+-float)
E₁ + E₂ : float

if (x == 3) {x := y + z;} else {a := b + c;}

if (x == 3) {x := y + z;} else {a := b + c;}: Stmt
Overloading

(+) Increase flexibility in programming

• Examples are when user wants to use an operator to express similar ideas.

• Example:

```c
int a, b, c;
int p[10], q[10], r[10];
int x[10][10], y[10][10], z[10][10];
a = b * c; // integer multiplication
p = a * q; // Scalar multiplication
x = y * z; // Matrix multiplication
```

• Therefore overloading is good.
Polymorphism – Overloading

Overloading

(−) Decrease Reliability (error detection)

• Examples are when user intends to use the operator in one context, but accidentally uses it in another.

• Example
  – In many languages, the minus sign is overloaded to both unary and binary uses.

\[ x = z - y \] and \[ x = -y \]

will both compile. What if I intend to do the first, but accidentally leave out the ‘z’?
Polymorphism – Overloading

Overloading

• Do you allow the user to perform overloading? (Flexibility) Or are all overloaded functions predefined in the language? (controlled reliability)

• If you allow the user to perform overloading, then can the user overload existing operators in the language?