Type Analysis

Is an operator applied to an “incompatible” operand?
Type checking:

- **Static**: Check for type compatibility at compile time
- **Dynamic**: Check for type compatibility at run time
Type analysis phase also used to resolve fields in a structure:

  Example: `list.element`
Type Checking vs. Type Inference

- A **Type Checker** only verifies that the given declarations are consistent with their use.
  Examples: type checkers for Pascal, C.
- A **Type Inference** system generates consistent type declarations from information implicit in the program.
  Examples: Type inference in SML, Scheme.

Given $y = 3.1415 \times x \times x$, we can infer that $y$ is a float.

Why Static Type Checking?

- Catch errors at compile time instead of run time.
- Determine which operators to apply.
  Example: In $x + y$, “+” is integer addition if $x$ and $y$ are both integers.
- Recognize when to convert from one representation to another (**Type Coercion**).
  Example: In $x + y$, if $x$ is a float while $y$ is an integer, convert $y$ to a float value before adding.
Type Checking: An Example

\[
E \rightarrow \text{int\_const} \quad \{ \ E.type = \text{int}; \ \}
\]
\[
E \rightarrow \text{float\_const} \quad \{ \ E.type = \text{float}; \ \}
\]
\[
E \rightarrow E_1 + E_2
\]
\[
\quad \{ \text{if } E_1.type == E_2.type == \text{int} \}
\]
\[
\quad \quad E.type = \text{int};
\]
\[
\quad \text{else}
\quad \quad E.type = \text{float};
\]

Type Checking: Another Example

\[
E \rightarrow \text{int\_const} \quad \{ \ E.type = \text{int}; \ \}
\]
\[
E \rightarrow \text{float\_const} \quad \{ \ E.type = \text{float}; \ \}
\]
\[
E \rightarrow \text{id} \quad \{ \ E.type = \text{sym\_lookup(id.entry, type);} \ \}
\]
\[
E \rightarrow E_1 + E_2
\]
\[
\quad \{ \text{if } (E_1.type \not\in \{\text{int, float}\}) \text{ OR } \}
\quad \quad (E_2.type \not\in \{\text{int, float}\})
\quad \quad E.type = \text{error};
\]
\[
\quad \text{else if } E_1.type == E_2.type == \text{int}
\quad \quad E.type = \text{int};
\]
\[
\quad \text{else}
\quad \quad E.type = \text{float};
\]
Types

- Base types: atomic types with no internal structure. Examples: int, char.
- Structured types: Types that combine (collect together) elements of other types.
  - Arrays: Characterized by dimensions, index range in each dimension, and type of elements.
  - Records: (structs and unions) Characterized by fields in the record and their types.

Type Expressions

Language to define types.

\[
\text{Type} \quad \rightarrow \quad \text{int} \mid \text{float} \mid \text{char} \ldots \\
\quad \quad \mid \text{void} \\
\quad \quad \mid \text{error} \\
\quad \quad \mid \text{name} \\
\quad \quad \mid \text{array}(\text{Type}) \\
\quad \quad \mid \text{record}( (\text{name}, \text{Type})^* ) \\
\quad \quad \mid \text{pointer}(\text{Type}) \\
\quad \quad \mid \text{tuple}( (\text{Type})^* ) \\
\quad \quad \mid \text{arrow}( \text{Type}, \text{Type} )
\]
Examples of Type Expressions

- float xform[3][3];
  xform ∈ array(array(float))
- char *string;
  string ∈ pointer(char)
- struct list { int element; struct list *next; } l;
  list ≡ record((element, int), (next, pointer(list)))
  l ∈ list
- int max(int, int);
  max ∈ arrow(tuple(int, int), int)

Type Checking with Type Expressions

\[
E \rightarrow E_1 [E_2] \quad \{ \text{if } E_1.\text{type} == \text{array}(T) \text{ AND} \}
\]
\[
E_2.\text{type} == \text{int} 
\]
\[
E.\text{type} = T \
\]
else
\[
E.\text{type} = \text{error} \}
\]
\[
E \rightarrow *E_1 \quad \{ \text{if } E_1.\text{type} == \text{pointer}(T) \}
\]
\[
E.\text{type} = T \
\]
else
\[
E.\text{type} = \text{error} \}
\]
\[
E \rightarrow &E_1 \quad \{ E.\text{type} = \text{pointer}(E_1.\text{type}) \} \]
Functions and Operators

Functions and Operators have *Arrow* types.

- \( \text{max}: \, \text{int} \times \text{int} \rightarrow \text{int} \)
- \( \text{sort}: \, \text{numlist} \rightarrow \text{numlist} \)

Functions and operators are *applied* to operands.

- \( \text{max}(x, y): \)

  \[
  \begin{align*}
  \text{max} & : \, \text{int} \times \text{int} \rightarrow \text{int} \\
  x & : \, \text{int} \\
  y & : \, \text{int} \\
  (x, y) & : \, \text{int} \times \text{int} \\
  \text{max}(x, y) & : \, \text{int}
  \end{align*}
  \]

Function Application

\[
\begin{align*}
E & \rightarrow \ E_1 \ E_2 & \{ \text{if } E_1.\text{type} \equiv \text{arrow}(S, T) \text{ AND } \\
 & \quad E_2.\text{type} \equiv S \} \\
& \text{else} \\
& \quad E.\text{type} = \text{error} \}
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow \ ( E_1, E_2 ) & \{ \text{E.type = tuple}(E_1.\text{type}, E_2.\text{type}) \}
\end{align*}
\]
Type Equivalence

When are two types “equal”? 

```plaintext
type Vector = array [1..10] of real;
type Weights = array [1..10] of real;

var x, y: Vector;
z: Weight;
```

• **Name Equivalence:** When they have the same name. 
  x and y have same type, but z has different type.

• **Structural Equivalence:** When they have the same structure. 
  x, y and z have same type.

Structural Equivalence

**S ≡ T** iff:

- S and T are the same **basic type**;
- S = `array(S_1)` , T = `array(T_1)`, and $S_1 \equiv T_1$.
- S = `pointer(S_1)` , T = `pointer(T_1)`, and $S_1 \equiv T_1$.
- S = `tuple(S_1, S_2)` , T = `tuple(T_1, T_2)`, and $S_1 \equiv T_1$ and $S_2 \equiv T_2$.
- S = `arrow(S_1, S_2)` , T = `arrow(T_1, T_2)`, and $S_1 \equiv T_1$ and $S_2 \equiv T_2$. 
Subtyping

Object-oriented languages permit subtyping.

    class Rectangle {
        private int x,y;
        int area() { ... }
    }
    class Square extends Rectangle {
        ...
    }

Square is a subclass of Rectangle.
Since all methods on Rectangle are inherited by Square (unless explicitly overridden)

      Square is a subtype of Rectangle.

Inheritance

    class Circle {
        float x, y; // center
        float r; // radius
        float area() {
            return 3.1415 * r * r;
        }
    }

    class ColoredCircle extends Circle {
        Color c;
    }

    class Test{
        static main() {
            ColoredCircle t;
            ... t.area() ...
        }
    }
Resolving Names

What entity is represented by $t.area()$?
(assume no overloading)

- Determine the type of $t$.
  $t$ has to be of type $\text{user}(c)$.
- If $c$ has a method of name $\text{area}$, we are done.
  Otherwise, if the superclass of $c$ has a method of name $\text{area}$, we are done.
  Otherwise, if the superclass of superclass of $c$...

  $\Rightarrow$ Determine the least $\text{superclass}$ of class $c$ that has a method with name $\text{area}$.

Overloading

class Rectangle {
    int x,y; // top lh corner
    int l, w; // length and width

    Rectangle move() {
        x = x + 5; y = y + 5;
        return this;
    }

    Rectangle move(int dx, int dy) {
        x = x + dx; y = y + dy;
        return this;
    }
}
Resolving Overloaded Names

What entity is represented by `move` in `r.move(3, 10)`?

- Determine the type of `r`.
  - `r` has to be of type `user(c)`.
- Determine the nearest `superclass` of class `c` that has a method with name `move` such that `move` is a method that takes two `int` parameters.

Structural Subtyping

\[ S \subseteq T \text{ iff:} \]

- `S` and `T` are the same basic type.
- `S = user(type_1), T = user(type_2)` and `type_1 \subseteq type_2`.
- `S = array(S_1), T = array(T_1)`, and `S_1 \subseteq T_1`;
- `S = pointer(S_1), T = pointer(T_1)`, and `S_1 \subseteq T_1`;
- `S = tuple(S_1, S_2), T = tuple(T_1, T_2)`, and `S_1 \subseteq T_1` and `S_2 \subseteq T_2`;
- `S = arrow(S_1, S_2), T = arrow(T_1, T_2)`, and \( S_1 \supseteq T_1 \) and `S_2 \subseteq T_2`. 
Inheritance and Overloading

What entity is represented by \( f \) in \( E.f(a_1, a_2, \ldots, a_n) \)?

- Let the type of \( E \) be \( \text{user}(c) \).
- The target signature of \( f \) is \( \text{type}(a_1) \times \cdots \times \text{type}(a_n) \rightarrow ? \).
  (Call target signature as \( T \))
- The selected method is the one defined in the least superclass of class \( c \) such that type of the method is a \textit{subtype} of \( T \).
- If there are multiple methods in a superclass of \( c \), say \( f_1, f_2, \ldots, f_n \) with signatures \( T_1, T_2, \ldots T_n \) respectively,
- ... select \( f_i \) such that \( T_i \) is the (unique) greatest type such that \( T_i \subseteq T \).

Inheritance: Another Example
Abstract objects and Concrete Representations

Abstract classes *declare* methods, but do not *define* them.

**Example:**
- `closed_graphical` declares “area” method, but cannot define the method.
- The different “area” methods are defined when the object’s representations are concrete: in rectangle, ellipse, etc.

When “area” method is applied to an object of class `closed_graphical`, we method to be called is the one defined in rectangle, triangle, ellipse, etc. ... which can be resolved only at run-time!

Types in OO Languages: The Whole Story

Decaf implements a small part of the type system for an OO language.

- **Subtype rule:** Wherever an object of type `t` is required (as a parameter of a method, return value, or rhs of assignments), object of any subtype `s` of `t` can be used.
Method Selection rule: If class B inherits from class A and overwrites method m, then for any B object b, method m of B must be used, even if b is used as an A object.

```
class A {
    int m() { ... }
}
class B extends A {
    int m() { ... }
}
class C{
    int f(B b) {
        A a;
        a = b;
        ... a.m() ...
    }
}
```

Dynamic Binding rule: A method of object obj, which can be potentially overwritten in a subclass has to be bound dynamically if the compiler cannot determine the runtime type of obj.