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

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

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

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.

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

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

\[ E \rightarrow E_1[E_2] \{ \text{if } E_1.type == \text{array}(T) \text{ AND } E_2.type == \text{int} \]
\[ \quad E.type = T \]
\[ \text{else} \]
\[ \quad E.type = \text{error} \} \]

\[ E \rightarrow *E_1 \{ \text{if } E_1.type == \text{pointer}(T) \}
\[ \quad E.type = T \]
\[ \text{else} \]
\[ \quad E.type = \text{error} \} \]

\[ E \rightarrow &E_1 \{ E.type = \text{pointer}(E_1.type) \} \]

Functions and Operators

Functions and Operators have Arrow types.

- \textbf{max}: \textit{int} \times \textit{int} \rightarrow \textit{int}
- \textbf{sort}: \textit{numlist} \rightarrow \textit{numlist}

Functions and operators are applied to operands.

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

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

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

\[
E \rightarrow (E_1, E_2) \quad \{ E.\text{type} = \text{tuple}(E_1.\text{type}, E_2.\text{type}) \}
\]

Type Equivalence

When are two types “equal”? 

\[
\text{type Vector} = \text{array}[1..10] \text{ of real}; \\
\text{type Weights} = \text{array}[1..10] \text{ of real};
\]

\[
\text{var } x, y : \text{Vector}; \\
\text{z: Weight;}
\]

- **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

$S \equiv T$ iff:

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

### Subtyping

Object-oriented languages permit subtyping.

```java
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 \( t \text{.} \text{area}() \)?
(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 \( c \)...

\( \implies \) Determine the nearest \textit{superclass} of class \( c \) that has a method with name \text{area}. 
Overloading

```java
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

\( S \subseteq T \) iff:

- \( S \) and \( T \) are the same basic type.
- \( S = \text{user}(\text{type}_1) \), \( T = \text{user}(\text{type}_2) \) and \( \text{type}_1 \subseteq \text{type}_2 \).
- \( S = \text{array}(S_1) \), \( T = \text{array}(T_1) \) and \( S_1 \subseteq T_1 \);
- \( S = \text{pointer}(S_1) \), \( T = \text{pointer}(T_1) \) and \( S_1 \subseteq T_1 \);
- \( S = \text{tuple}(S_1, S_2) \), \( T = \text{tuple}(T_1, T_2) \), and \( S_1 \subseteq T_1 \) and \( S_2 \subseteq T_2 \);
- \( S = \text{arrow}(S_1, S_2) \), \( T = \text{arrow}(T_1, T_2) \), and \( S_1 \subseteq T_1 \) and \( T_2 \equiv S_2 \).

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 \( \text{user}(c) \).
- \( f \) is the method in the nearest superclass of class \( c \) such that type of \( f \) is a supertype of \( \text{type}(a_1) \times \cdots \text{type}(a_1) \rightarrow \perp \).
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!
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 \( b \) was used as an \( A \) object.

```java
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.