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 * x * 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

$$
\begin{array}{lcll}
E & \longrightarrow & \texttt{int\_const} & \{\ E.type = int;\ \} \\
E & \longrightarrow & \texttt{float\_const} & \{\ E.type = float;\ \} \\
E & \longrightarrow & E_1 \texttt{ + } E_2 & \{ \\
 & & & \text{if } E_1.type == E_2.type == int \\
 & & & \qquad E.type = int; \\
 & & & \text{else} \\
 & & & \qquad E.type = float; \\
 & & & \}
\end{array}
$$

Type Checking: Another Example

$$
\begin{array}{lcll}
E & \longrightarrow & \texttt{int\_const} & \{\ E.type = int;\ \} \\
E & \longrightarrow & \texttt{float\_const} & \{\ E.type = float;\ \} \\
E & \longrightarrow & \texttt{id} & \{\ E.type = sym\_lookup(\texttt{id}.entry, \texttt{type});\ \} \\
E & \longrightarrow & E_1 \texttt{ + } E_2 & \{ \\
 & & & \text{if } (E_1.type \notin \{int, float\}) \text{ OR} \\
 & & & \qquad\qquad (E_2.type \notin \{int, float\}) \\
 & & & \qquad E.type = error; \\
 & & & \text{else if } E_1.type == E_2.type == int \\
 & & & \qquad E.type = int; \\
 & & & \text{else} \\
 & & & \qquad E.type = float; \\
 & & & \}
\end{array}
$$

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.

$$
\begin{array}{lcl}
\mathit{Type} & \longrightarrow & \texttt{int} \mid \texttt{float} \mid \texttt{char} \ldots \\
 & \mid & \texttt{void} \\
 & \mid & \texttt{error} \\
 & \mid & \texttt{name} \\
 & \mid & \texttt{array}(\texttt{size}, \mathit{Type}) \\
 & \mid & \texttt{record}(\ (\texttt{name}, \mathit{Type})^* \ ) \\
 & \mid & \texttt{pointer}(\mathit{Type}) \\
 & \mid & \texttt{tuple}(\ (\mathit{Type})^* \ ) \\
 & \mid & \texttt{arrow}(\ \mathit{Type}, \mathit{Type})
\end{array}
$$

Examples of Type Expressions

- `float xform[3][3];`
  $\texttt{xform} \in \texttt{array}(3, \texttt{array}(3, \texttt{float}))$
- `char *string;`
  $\texttt{string} \in \texttt{pointer}(\texttt{char})$
- `struct list { int element; struct list *next; } l;`
  $\mathit{list} \equiv \texttt{record}(\ (\mathit{element}, \texttt{int}), (\mathit{next}, \texttt{pointer}(\mathit{list})\ ))$
  $\texttt{l} \in \mathit{list}$
- `int max(int, int);`
  $\texttt{max} \in \texttt{arrow}(\ \texttt{tuple}(\ \texttt{int}, \texttt{int}), \texttt{int})$

Type Checking with Type Expressions

$$
\begin{array}{lcll}
E & \longrightarrow & E_1 \; \texttt{[} \; E_2 \; \texttt{]} & \{ \text{ if } E_1.\mathit{type} == \texttt{array}(\mathbf{S}, \mathbf{T}) \text{ AND} \\
 & & & \qquad\qquad E_2.\mathit{type} == \texttt{int} \\
 & & & \qquad E.\mathit{type} = \mathbf{T} \\
 & & & \text{else} \\
 & & & \qquad E.\mathit{type} = \texttt{error} \; \} \\
E & \longrightarrow & \texttt{*} \; E_1 & \{ \text{ if } E_1.\mathit{type} == \texttt{pointer}(\mathbf{T}) \\
 & & & \qquad E.\mathit{type} = \mathbf{T} \\
 & & & \text{else} \\
 & & & \qquad E.\mathit{type} = \texttt{error} \; \} \\
E & \longrightarrow & \texttt{\&} \; E_1 & \{ \; E.\mathit{type} = \texttt{pointer}(E_1.\mathit{type}) \; \}
\end{array}
$$

Functions and Operators

Functions and Operators have *Arrow* types.

- `max`: $\mathit{int} \times \mathit{int} \longrightarrow \mathit{int}$
- `sort`: $\mathit{numlist} \longrightarrow \mathit{numlist}$

Functions and operators are *applied* to operands.

- `max(x,y)`:

$$
\begin{array}{rcl}
\texttt{max} & : & \mathit{int} \times \mathit{int} \longrightarrow \mathit{int} \\
\texttt{x} & : & \mathit{int} \\
\texttt{y} & : & \mathit{int} \\
(\texttt{x}, \texttt{y}) & : & \mathit{int} \times \mathit{int} \\
\texttt{max}(\texttt{x}, \texttt{y}) & : & \mathit{int}
\end{array}
$$

Function Application

$$
\begin{array}{llll}
E & \longrightarrow & E_1\ E_2 & \{\ \text{if } E_1.type \equiv \texttt{arrow}(\mathbf{S},\ \mathbf{T}) \text{ AND} \\
 & & & \qquad\qquad E_2.type \equiv \mathbf{S} \\
 & & & \qquad E.type = \mathbf{T} \\
 & & & \text{else} \\
 & & & \qquad E.type = \texttt{error}\ \} \\
E & \longrightarrow & (\ E_1,\ E_2\ ) & \{\ E.type = \texttt{tuple}(E_1.type,\ E_2.type)\ \}
\end{array}
$$

Type Equivalence

When are two types "equal"?

```
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 $\equiv$ **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 `user`(c).
- If c has a method of name `area`, we are done.
  Otherwise, if the superclass of c has a method of name `area`, we are done.
  Otherwise, if the superclass of superclass of c...

$\Longrightarrow$ Determine the nearest *superclass* of class c that has a method with name `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** iff:

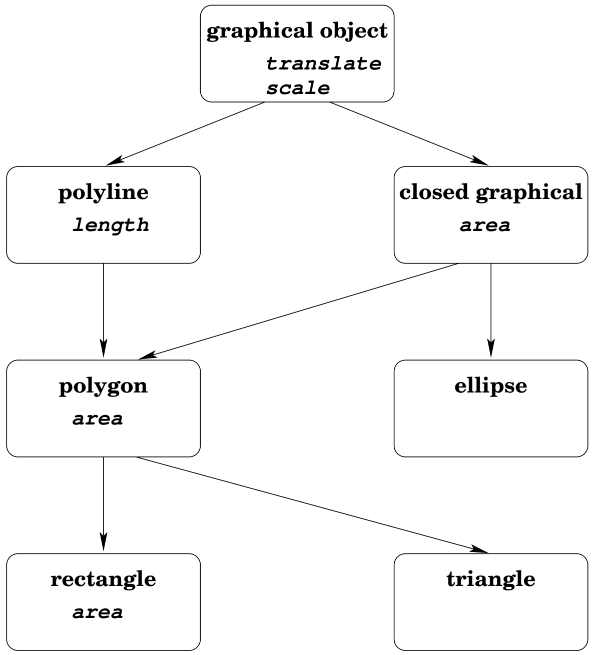
- **S** and **T** are the same **basic type**.
- **S** $=$ `user`$(type_1)$, **T** $=$ `user`$(type_2)$ and $type_1 \subseteq type_2$.
- **S** $=$ `array`(S_1) , **T** $=$ `array`(T_1), and $S_1 \subseteq T_1$;
- **S** $=$ `pointer`(S_1) , **T** $=$ `pointer`(T_1), and $S_1 \subseteq T_1$;
- **S** $=$ `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 $S_1 \subseteq T_1$ and $T_2 \equiv S_2$.

Inheritance and Overloading

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

- Let the type of E be `user`(c).
- `f` is the method in the nearest superclass of class c such that type of `f` is a <u>*supertype*</u> of $type(a_1) \times \cdots type(a_1) \rightarrow \perp$.

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.

Types in OO Languages: The Whole Story (contd.)

- **Method Selection rule:** If class `B` inherits from class tt A and overwrites method `m`, then for any `B` object b, method `m` of `B` must be used, even if b us 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() ...
}
```

Types in OO Languages: The Whole Story (contd.)

- **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*.