

Types, Type Checking and Type Inference

Compiler Design

CSE 504

Last modified: Mon Mar 21 2016 at 12:30:21 EDT
Version: 1.3 20:58:09 2012/04/09
Compiled at 12:31 on 2016/03/21

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

```
 $E \longrightarrow \text{int\_const} \quad \{ E.type = \text{int}; \}$   
 $E \longrightarrow \text{float\_const} \quad \{ E.type = \text{float}; \}$   
 $E \longrightarrow E_1 + E_2 \quad \{$   
    if  $E_1.type == E_2.type == \text{int}$   
         $E.type = \text{int};$   
    else  
         $E.type = \text{float};$   
    }
```

Type Checking: Another Example

```
 $E \longrightarrow \text{int\_const} \quad \{ E.type = \text{int}; \}$   
 $E \longrightarrow \text{float\_const} \quad \{ E.type = \text{float}; \}$   
 $E \longrightarrow \text{id} \quad \{ E.type = \text{sym\_lookup}(\text{id.entry}, \text{type}); \}$   
 $E \longrightarrow E_1 + E_2 \quad \{$   
    if ( $E_1.type \notin \{\text{int}, \text{float}\}$ ) OR  
        ( $E_2.type \notin \{\text{int}, \text{float}\}$ )  
         $E.type = \text{error};$   
    else if  $E_1.type == E_2.type == \text{int}$   
         $E.type = \text{int};$   
    else  
         $E.type = \text{float};$   
     $\}$ 
```

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.

Type $\longrightarrow$ `int` | `float` | `char` ...
|
`void`
|
`error`
|
`name`
|
`array`(*Type*)
|
`record`((`name`, *Type*)*)
|
`pointer`(*Type*)
|
`tuple`((*Type*)*)
|
`arrow`(*Type*, *Type*)

Examples of Type Expressions

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

Type Checking with Type Expressions

$$\begin{aligned} E &\longrightarrow E_1 [E_2] && \{ \text{if } E_1.type == \text{array}(\mathbf{T}) \text{ AND} \\ &&& \quad E_2.type == \text{int} \\ &&& \quad E.type = \mathbf{T} \\ &&& \text{else} \\ &&& \quad E.type = \text{error} \} \\ E &\longrightarrow * E_1 && \{ \text{if } E_1.type == \text{pointer}(\mathbf{T}) \\ &&& \quad E.type = \mathbf{T} \\ &&& \text{else} \\ &&& \quad E.type = \text{error} \} \\ E &\longrightarrow \& E_1 && \{ E.type = \text{pointer}(E_1.type) \} \end{aligned}$$

Functions and Operators

Functions and Operators have *Arrow* types.

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

Functions and operators are *applied* to operands.

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

$$\begin{array}{lll} \text{max} & : & \text{int} \times \text{int} \longrightarrow \text{int} \\ \text{x} & : & \text{int} \\ \text{y} & : & \text{int} \\ (\text{x}, \text{y}) & : & \text{int} \times \text{int} \\ \text{max}(\text{x}, \text{y}) & : & \text{int} \end{array}$$

Function Application

$$E \longrightarrow E_1 E_2 \quad \left\{ \begin{array}{l} \text{if } E_1.type \equiv \text{arrow}(\mathbf{S}, \mathbf{T}) \text{ AND} \\ \quad E_2.type \equiv \mathbf{S} \\ \quad E.type = \mathbf{T} \\ \text{else} \\ \quad E.type = \text{error} \end{array} \right\}$$
$$E \longrightarrow (E_1, E_2) \quad \{ E.type = \text{tuple}(E_1.type, E_2.type) \}$$

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**;
- $S = \text{array}(S_1)$, $T = \text{array}(T_1)$, 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.

```
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`...
 - ⇒ Determine the least 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**.
- $S = \text{user}(type_1)$, $T = \text{user}(type_2)$ and $type_1 \subseteq type_2$.
- $S = \text{array}(S_1)$, $T = \text{array}(T_1)$, 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 $S_1 \supseteq T_1$ and $S_2 \subseteq T_2$.

Inheritance and Overloading

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

- Let the type of E be `user(c)`.

Inheritance and Overloading

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

- Let the type of E be `user(c)`.
- The *target* signature of f is $type(a_1) \times \dots \times type(a_n) \rightarrow ?$.
(Call target signature as T)

Inheritance and Overloading

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

- Let the type of E be `user`(c).
- The *target* signature of f is $\text{type}(a_1) \times \dots \times \text{type}(a_n) \rightarrow ?$.
(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 **subtype** of T .

Inheritance and Overloading

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

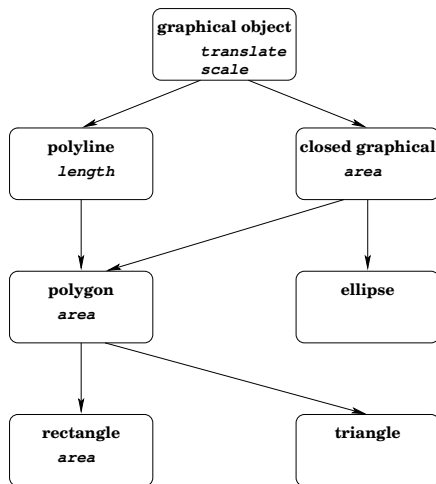
- Let the type of E be `user(c)`.
- The *target* signature of f is $type(a_1) \times \dots \times type(a_n) \rightarrow ?$.
(Call target signature as T)
- The selected method is the one defined in the least superclass of class c such that type of the method is a **subtype** of T .
- If there are multiple methods in a superclass of c , say $f_1, f_2, \dots, f_n$ with signatures $T_1, T_2, \dots, T_n$ respectively,

Inheritance and Overloading

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

- Let the type of E be `user`(c).
- The *target* signature of f is $type(a_1) \times \dots \times type(a_n) \rightarrow ?$.
(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 **subtype** of T .
- If there are multiple methods in a superclass of c , say $f_1, f_2, \dots, f_n$ with signatures $T_1, T_2, \dots, T_n$ 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.

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

- **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.

<pre>class A { int m() { ... } }</pre>	<pre>class B extends A { int m() { ... } }</pre>
--	--

```
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*.