University of Puerto Rico
Mayagüez Campus
College of Engineering
Department of Electrical and Computer Engineering

ICOM4029 – Compilers
Professor: Bienvenido Vélez
Technical Assistant: René D. Badía

Laboratory 8 – Semantic Analysis (continued)

I. Role of the Semantic Analyzer

The semantic analyzer should perform the following checks:

1. All identifiers are declared
2. Types
3. Inheritance relationships
4. Classes defined only once
5. Methods in a class defined only once
6. Reserved identifiers (like self) are not misused

The semantic analyzer should attempt to recover from errors and continue. A simple recovery mechanism is to assign the type Object to any expression that cannot otherwise be given a type.

II. Names and Scope Elements

- Class names introduced by class declarations
- Method names introduced by method definitions
- Object Id's introduced by:
  - Let expressions
  - Formal parameters
  - Attribute definitions in a class
  - Branches of case expressions

Notes:

- Attribute names are global within the class in which they are defined and classes that inherit from it.
- Use of a variable refers to the closest enclosing binding in the execution of the program.
- Methods and attributes may have the same name. (i.e. they can co-exist in the same scope)

III. Methods

1. Notes

- A method need not be defined in the class in which it is used, but in some parent class.
- Methods may be redefined (overridden). But, the redefined method should have the same formal parameters (with the same types) and the same return type as the original.

2. Method Dispatch

- Information about the formal parameters of the method must be known apriori. (May be in the current class or a parent class)

a) Normal Dispatch

The method must be defined on the inferred type class (or a parent) of the dispatch expression.

For example, if class B inherits A:

```
…
x : A <- new B;
…
x.foo(par)
…
```

`foo` must be defined in class A or a parent of A.
Also note that the inferred type of argument `par` must be less or equal to the declared type of `foo`'s formal parameter.
`foo(par)` without a dispatch expression indicates that it is a dispatch to self.
In other words, it is equivalent to using: `self.foo(par)`

b) Static Dispatch
It differs from the normal dispatch in that the method is found in the class explicitly named by the programmer.
The inferred type of the dispatch expression must conform to the specified type.
e.g.
`x@A.foo(par)`

Here, the inferred type of x must be ≤ A.

IV. SELF_TYPE

1. Definition

If SELF_TYPE appears textually in the class C as the declared type of E then it denotes the dynamic type of the "self" expression:

`dynamic_type(E) = dynamic_type(self) ≤ C`

The meaning of SELF_TYPE depends on where it appears. SELF_TYPE_C refers to an occurrence of SELF_TYPE in the body of C
The rule $\text{SELF\_TYPE}_c \leq C$ allows us to replace C in place of SELF_TYPE when we are type-checking, in most cases.

2. SELF_TYPE effect on type rules

New:

$$\frac{T' = \begin{cases} \text{SELF\_TYPE}_C & \text{if } T = \text{SELF\_TYPE} \\ T & \text{otherwise} \end{cases}}{O, M, C \vdash new\ T : T'}$$

Let-Init:

$$\frac{\begin{array}{l} T_0' = \begin{cases} \text{SELF\_TYPE}_C & \text{if } T_0 = \text{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0' \\ O[T_0'/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{let } x : T_0 \leftarrow e_1 \text{ in } e_2 : T_2}$$

Let-No-Init:

$$\frac{T_0' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \qquad O[T_0'/x], M, C \vdash e_1 : T_1}{O, M, C \vdash \text{let } x : T_0 \text{ in } e_1 : T_1}$$

Dispatch:

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0' = \begin{cases} C & \text{if } T_0 = \texttt{SELF\_TYPE}_C \\ T_0 & \text{otherwise} \end{cases} \\ M(T_0', f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T_{n+1}' = \texttt{SELF\_TYPE} \\ T_{n+1}' & \text{otherwise} \end{cases} \end{array}}{O, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}}$$

Static Dispatch:

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0 \leq T \\ M(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T_{n+1}' = \texttt{SELF\_TYPE} \\ T_{n+1}' & \text{otherwise} \end{cases} \end{array}}{O, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}}$$

Within a class `C` if `x` is declared as `x : T`, then:

$$O_C(x) = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T = \texttt{SELF\_TYPE} \\ T & \text{otherwise} \end{cases}$$

Attr-Init:

$$\frac{\begin{array}{l} O_C(x) = T_0 \\ O_C[\texttt{SELF\_TYPE}_C/self], M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C, M, C \vdash x : T_0 \leftarrow e_1;}$$

Attr-No-Init:

$$\frac{O_C(x) = T}{O_C, M, C \vdash x : T;}$$

Method:

$$\frac{\begin{array}{l} M(C, f) = (T_1, \ldots, T_n, T_0) \\ O_C[\texttt{SELF\_TYPE}_C/self][T_1/x_1] \ldots [T_n/x_n], M, C \vdash e : T_0' \\ T_0' \leq \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \end{array}}{O_C, M, C \vdash f(x_1 : T_1, \ldots, x_n : T_n) : T_0 \ \{ \ e \ \};}$$