Compiler Construction

Further with type checking

Goal

How to handle objects?

Goal

How to handle objects?

We need more definitions!

Subtyping

The notation $X \leq Y$ means:

1. X is a sub class of Y
2. X comforts to Y
3. An object of type X can be used when an object of type Y would have been acceptable

Subtyping

The notation $X \leq Y$ means:

1. X is a sub class of Y
2. X comforts to Y
3. An object of type X can be used when an object of type Y would have been acceptable

Remark on transitivity

$$X \leq Z \text{ and } Z \leq Y \implies X \leq Y$$

Robustness Theorem

Robustness Theorem

$\forall E$, dynamic_type(E) $\leq$ static_type(E)

In most OO languages:

- Sub classes can only add more attributes or methods
- Methods can be redefined but only with same types

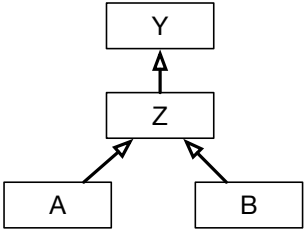
Not true in Eiffel

Problem statement

Consider the rule.

$$\frac{\Gamma \vdash \mathrm{c} : \mathrm{int} \qquad \Gamma \vdash e_1 : \mathrm{A} \qquad \Gamma \vdash e_2 : \mathrm{B}}{\Gamma \vdash \text{if c then } e_1 \text{ else } e_2 \text{: ???}}$$

And this inheritance diagram.



Least Upper Bound

Least Upper Bound (LUB)

LUB(X,Y) denotes the least upper bound to X and Y.

LUB(X,Y) = Z iff

- $X \leq Z$ and $Y \leq Z$ (Z is a super class)
- and $X \leq Z'$ and $Y \leq Z' \implies Z \leq Z'$

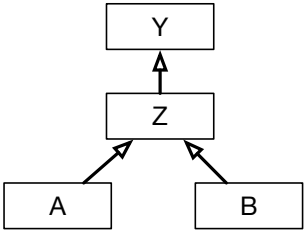
Compute static information.

Rewrite the if rule

Generalization.

$$\frac{\Gamma \vdash \text{c} : \text{int} \qquad \Gamma \vdash e_1 : \text{A} \qquad \Gamma \vdash e_2 : \text{B}}{\Gamma \vdash \text{if c then } e_1 \text{ else } e_2 \text{: LUB(A, B)}}$$

With LUB(A,B) = Z.



How to implement $\leq$?

From / To	Class Type	Primitive Type	Array Type	Null Type	Error Type
Class Type	if same or inherits	No	No	No	No
Primitive Type	No	if same	No	No	No
Array Type	No	No	if underlying types match	No	No
Null Type	Yes	No	No	Yes	No
Error Type	Yes	Yes	Yes	Yes	Yes

Deeper inside overloading

What about overloading?

Simple overloading (without inheritance)

Life is simple!

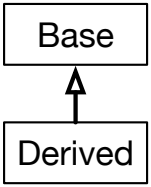
1. Consider all overloaded functions
2. Filter unsuitable functions
3. If exactly one function remains: pick it!
4. Otherwise (none or more than one function) report error

Overloading with inheritance (1/7)

Consider these functions:

```
void foo (Base, Base);
void foo (Base, Derived);
void foo (Derived, Derived);
```

With this hierarchy:



Overloading with inheritance (2/7)

```
void foo (Base, Base);
void foo (Base, Derived);
void foo (Derived, Derived);
```

How to handle

```
foo (new Derived(),
     new Derived());
```

Goal

Find the best specialization!

Overloading with inheritance (3/7)

```
void foo (Base, Base);
void foo (Base, Derived);
void foo (Derived, Derived);
```

How to handle

```
foo (new Base(),
     new Derived());
```

Goal

Find the best specialization!

Overloading with inheritance (4/7)

```
void foo (Base, Base);
void foo (Base, Derived);
void foo (Derived, Derived);
```

How to handle

```
foo (new Derived(),
     new Base());
```

Goal

Find the best specialization!

Overloading with inheritance (5/7)

```
void foo (Base, Base);
void foo (Base, Derived);
void foo (Derived, Base);
```

How to handle

```
foo (new Base(),
     new Base());
```

Goal

Find the best specialization!

Overloading with inheritance (6/7)

```
void foo (Base, Base);
void foo (Base, Derived);
void foo (Derived, Base);
```

How to handle

```
foo (new Derived(),
     new Derived());
```

Goal

Find the best specialization!

Overloading with inheritance (6/7)

Pick the best specialization!

Partial Ordering

- Consider 2 functions A and B
- A has form A(A_1, .., A_n)
- B has form B(B_1, .., B_n)
- $A \leq B$ iff $\forall i \in [1..n]$ $A_i \leq B_i$

If a best specialization exists, pick it
Otherwise the call is **ambiguous**

Overloading and variadic functions (1/2)

Consider these functions:

```
void foo (Base, Base);
void foo (Base, Derived);
void foo (Derived, ...);
```

Calling

```
foo (new Derived(),
     new Derived());
```

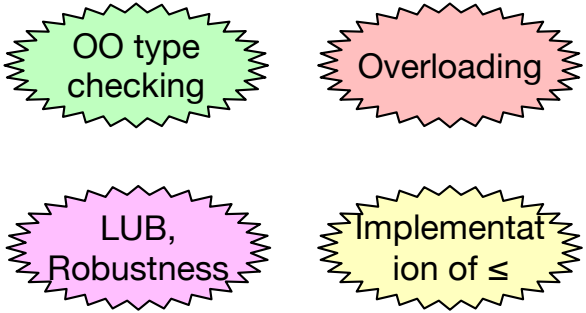
is either ambigous
or preference is given to non-variadic function (C++)

Overloading and variadic functions (2/2)

How to handle that?

1. Build a hierarchy with the set of candidates (functions)
2. Each level group functions of the same “abstraction” level
3. Start by the lower level
4. Filter unsuitable functions
5. If only one function remains pick it
6. If multiple functions remain : **ambiguous call**
7. Otherwise start over from the next level

Summary


OO type checking
Overloading
LUB, Robustness
Implementation of ≤