Inclusion Polymorphism

Consider the last two lines of the example in the following slide:

- In $L_1$, $S$ points to a `Shape` object, but it could just as well have pointed to an object of any one of `Shape`'s subtypes, `Square` and `Circle`.
- If, for example, $S$ had been a `Circle`, the assignment $C := S$ would have been perfectly OK. In $L_2$, however, $S$ is a `Shape` and the assignment $C := S$ is illegal (a `Shape` isn't a `Circle`).

Typechecking Rules

A variable of type `T` may refer to an object of `T` or one of `T`'s subtypes.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Compile-time</th>
<th>Run-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t := r$</td>
<td>Legal</td>
<td>Legal</td>
</tr>
<tr>
<td>$t := u$</td>
<td>Legal</td>
<td>Legal</td>
</tr>
<tr>
<td>$u := t$</td>
<td>Legal</td>
<td>Check</td>
</tr>
<tr>
<td>$s := u$</td>
<td>Illegal</td>
<td>Illegal</td>
</tr>
</tbody>
</table>

\[1\]

Inclusion Polymorphism

VAR $S : \text{Shape}; \quad Q : \text{Square}; \quad C : \text{Circle};$
BEGIN
\quad Q := \text{NEW} \ (\text{Square});$
\quad C := \text{NEW} \ (\text{Circle});$
\quad S := Q; \quad (*) \ OK (*)$
\quad S := C; \quad (*) \ OK (*)$
\quad Q := C; \quad (*) \ Compile-time \ Error (*)$
\quad L_1: \quad S := \text{NEW} \ (\text{Shape});$
\quad L_2: \quad C := S; \quad (*) \ Run-time \ Error (*)$
END;
Run-time Type Checking

Modula-3 Type-test Primitives:
ISTYPE(object, T) Is object's type a subtype of T?
NARROW(object, T) If object's type is not a subtype of T, then issue a run-time type error. Otherwise return object, typecast to T.
TYPECASE Expr OF Perform different actions depending on the runtime type of Expr.

The assignment s := t is compiled into s := NARROW(t, TYPE(s)).

Run-time Type Checking...

The Modula-3 runtime-system has three functions that are used to implement typetests, casts, and the TYPECASE statement

NARROW takes a template and an object as parameter. It checks that the type of the object is a subtype of the type of the template. If it is not, a run-time error message is generated. Otherwise, NARROW returns the object itself.

1. ISTYPE(S,T : Template) : BOOLEAN;
2. NARROW(Object, Template) : Object;
3. TYPECODE(Object) : CARDINAL;

Run-time Checks

Casts are turned into calls to NARROW, when necessary:

VAR S : Shape; VAR C : Circle;
BEGIN
  S := NEW (Shape); C := S;
END;

VAR S : Shape; VAR C : Circle;
BEGIN
  S := malloc (SIZE(Shape));
  C := NARROW(S, Circle$Template);
END;

Implementing ISTYPE

We follow the object's template pointer, and immediately (through the templates' parent pointers) gain access to it's place in the inheritance hierarchy.

PROCEDURE ISTYPE (S, T : TemplatePtr) : BOOLEAN;
BEGIN
  LOOP
    IF S = T THEN RETURN TRUE; ENDIF;
    S := S^.parent;
    IF S = ROOT THEN RETURN FALSE; ENDIF;
  ENDLOOP
END ISTYPE;
Implementing NARROW

NARROW uses ISTYPE to check if $S$ is a subtype of $T$. Of so, $S$ is returned. If not, an exception is thrown.

PROCEDURE NARROW(T:TemplatePtr; S:Object):Object;
BEGIN
IF ISTYPE(S.$template, T) THEN
  RETURN S (* OK *)
ELSE WRITE "Type error"; HALT;
ENDIF;
END NARROW;

Run-time Checks — Example

TYPE T = CLASS [···];
S = T CLASS [···];
U = T CLASS [···];
V = U CLASS [···];
X = S CLASS [···];
Y = U CLASS [···];
Z = U CLASS [···];
VAR x : X;

Run-time Checks — Example...

The time for a type test is proportional to the depth of the inheritance hierarchy. Two algorithms do type tests in constant time:
1. Norman Cohen, “Type-Extension Type Tests can be Performed in Constant Time.”
2. Paul F. Dietz, “Maintaining Order in a Linked List”. The second is more efficient, but requires the entire type hierarchy to be known. This is a problem in separately compiled languages.

SRC Modula-3 uses Dietz’ method and builds type hierarchies of separately compiled modules at link-time.

These algorithms only work for single inheritance.

Run-time Checks – An $O(1)$ Algorithm
Run-time Checks – Alg. II (b)

In the Compiler (or Linker):
1. Build the inheritance tree.
2. Perform a preorder traversal and assign preorder numbers to each node.
3. Similarly, assign postorder numbers to each node.
4. Store T’s pre- and postorder numbers in T’s template.

In the Runtime System:

PROCEDURE ISTYPE ( S, T : TemplatePtr) : BOOLEAN;
BEGIN
RETURN (T.pre ≤ S.pre) AND (T.post ≥ S.post);
END ISTYPE;

Run-time Checks – Alg. II (c)

TYPE
T = CLASS [⋯];
S = T CLASS [⋯];
U = T CLASS [⋯];
V = U CLASS [⋯];
X = S CLASS [⋯];
Y = U CLASS [⋯];
Z = U CLASS [⋯];

√ISTYPE(Y, U)     U.pre ≤ Y.pre    U.post ≥ Y.post
ISTYPE(Z, S)     S.pre ≤ Z.pre    S.post ≥ Z.post
√ISTYPE(Z, T)     T.pre ≤ Z.pre    T.post ≥ Z.post

Run-time Checks – Alg. II (d)

Consider U:
1. U’s pre-number is ≤ all it’s children’s pre numbers.
2. U’s post-number is ≥ all it’s children’s post numbers.


Inlining Methods
Inlining Methods

Consider a method invocation \( m.P() \). The actual procedure called will depend on the run-time type of \( m \).

If more than one method can be invoked at a particular call site, we have to inline all possible methods. The appropriate code is selected code by branching on the type of \( m \).

To improve on method inlining we would like to find out when a call \( m.P() \) can call exactly one method.

Inlining Methods — Example

```
TYPE T = CLASS [f : T]
   METHOD M (); BEGIN END M;
];
TYPE S = CLASS EXTENDS T
   METHOD N (); BEGIN END N;
   METHOD M (); BEGIN END M;
];
VAR x : T; y : S;
BEGIN
   x.M();
   y.M();
END;
```

Type Hierarchy Analysis

For each type \( T \) and method \( M \) in \( T \), find the set \( S_{T,M} \) of method overrides of \( M \) in the inheritance hierarchy tree rooted in \( T \).

If \( x \) is of type \( T \), \( S_{T,M} \) contains the methods that can be called by \( x.M() \).

We can improve on type hierarchy analysis by using a variant of the Reaching Definitions data flow analysis.
Type Hierarchy Analysis...

```
TYPE T = CLASS [ ] [
  METHOD M (); BEGIN END M; ];
TYPE S = CLASS EXTENDS T [ ] [
  METHOD N (); BEGIN END N;
  METHOD M (); BEGIN END M; ];
VAR x : T; y : S;
BEGIN
  x.M(); ⇐ S\(_{T,M}\) = \{T.M, S.M\}
  y.M(); ⇐ S\(_{S,M}\) = \{S.M\}
END;
```

Readings and References

- Read Scott: 529–551, 554–561, 564–573
- The time for a type test is proportional to the depth of the inheritance hierarchy. Many algorithms do type tests in constant time:
  1. Norman Cohen, “Type-Extension Type Tests can be Performed in Constant Time.”
  2. Paul F. Dietz, “Maintaining Order in a Linked List”.

Confused Student Email

What happens when both a class and its subclass have an instance variable with the same name?

- The subclass gets both variables. You can get at both of them, directly or by casting. Here’s an example in Java:

```java
class C1 {int a;}
class C2 extends C1 {double a;}
class C {
  static public void main(String[] arg) {
    C1 x = new C1(); C2 y = new C2();
    x.a = 5; y.a = 5.5;
    ((C1)y).a = 5;
  }
}
```