

# CSc 520

## Principles of Programming Languages

### 19 : OO Languages — Polymorphism

Christian Collberg

collberg+520@gmail.com

Department of Computer Science  
University of Arizona

Copyright © 2008 Christian Collberg

—Spring 2008 — 19

[1]

# 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`).

520 —Spring 2008 — 19

[2]

## Inclusion Polymorphism

```
VAR S : Shape; Q : Square; C : Circle;
BEGIN
  Q := NEW (Square);
  C := NEW (Circle);

  S := Q; (* OK *)
  S := C; (* OK *)

  Q := C; (* Compile-time Error *)

  L1: S := NEW (Shape);
  L2: C := S; (* Run-time Error *)
END;
```

—Spring 2008 — 19

## Typechecking Rules

```
TYPE      T = CLASS ... END;
          U = T CLASS ... END;
          S = T CLASS ... END;
VAR       t,r : T; u : U; s : S;
```

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

Assignment	Compile-time	Run-Time
$t := r;$	Legal	Legal
$t := u;$	Legal	Legal
$u := t;$	Legal	Check
$s := u;$	Illegal	

520 —Spring 2008 — 19

# 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;
```

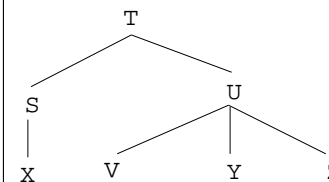
# Implementing NARROW

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

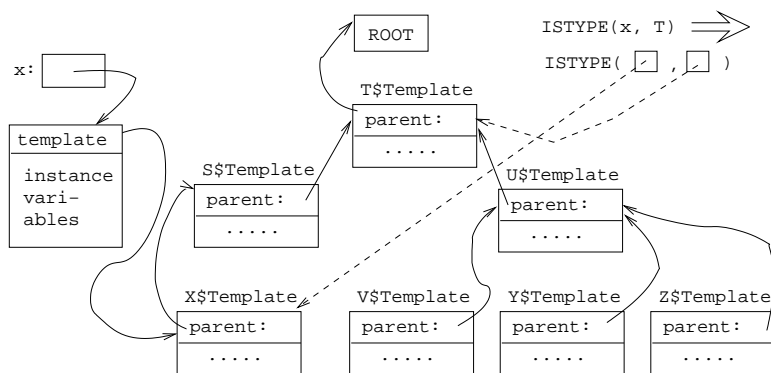
```
PROCEDURE NARROW(T:TemplatePtr; S:Object):Object;
BEGIN
  IF ISTYPE(S^.template, T) THEN
    RETURN S (* OK *)
  ELSE WRITE "Type error"; HALT;
  ENDIF;
END NARROW;
```

# 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...



# Run-time Checks – An $O(1)$ Algorithm

- 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 – Alg. II (b)

## In the Compiler (or Linker):

- Build the inheritance tree.
- Perform a preorder traversal and assign preorder numbers to each node.
- Similarly, assign postorder numbers to each node.
- Store T's pre- and postorder numbers in T's template.

## In the Runtime System:

```

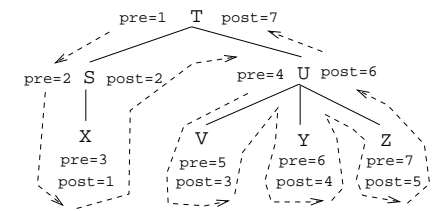
PROCEDURE ISTYPE (
  S, T : TemplatePtr) : BOOLEAN;
BEGIN
  RETURN (T.pre ≤ S.pre) AND (T.post ≥ S.post);
END ISTYPE;
    
```

# Run-time Checks – Alg. II (c)

## TYPE

```

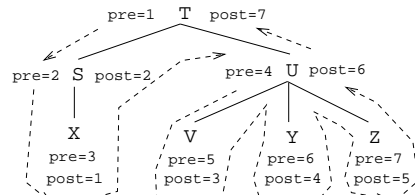
T = CLASS [... ];
S = T CLASS [... ];
U = T CLASS [... ];
V = U CLASS [... ];
X = S CLASS [... ];
Y = U CLASS [... ];
Z = U CLASS [... ];
    
```



$\sqrt{\text{ISTYPE}(Y, U)}$	$U.pre \leq Y.pre$	$U.post \geq Y.post$
$\text{ISTYPE}(Z, S)$	$S.pre \leq Z.pre$	$S.post \not\geq Z.post$
$\sqrt{\text{ISTYPE}(Z, T)}$	$T.pre \leq Z.pre$	$T.post \geq Z.post$

# Run-time Checks – Alg. II (d)

- Consider U:
  - U's pre-number is  $\leq$  all it's children's pre numbers.
  - U's post-number is  $\geq$  all it's children's post numbers.
 [U.pre, U.post] "covers" (in the sense that  $U.pre \leq pre$  and  $U.post \geq post$ ) the [pre, post] of all it's children.
- S is not a subtype of U since [U.pre, U.post] does not cover [S.pre, S.post] ( $S.post \leq U.post$  but  $S.pre \not\leq U.pre$ ).

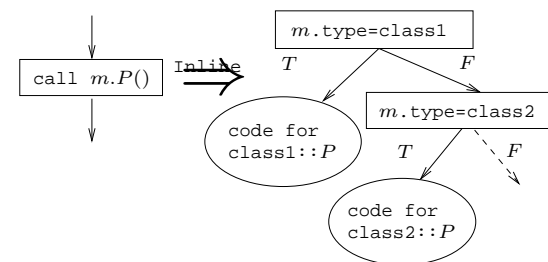


# Inlining Methods

# 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...



## 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();  $\Leftarrow S_{T,M} = \{T.M, S.M\}$   
    y.M();  $\Leftarrow S_{S,M} = \{S.M\}$   
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:

```
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;  
    }  
}
```