

# Polymorphism

## Lecture 25

Prof. Necula CS 164 Lecture 25

1

### Overview

---

- Why polymorphism ?
- Adding polymorphism to Cool
- Type checking
- Code generation
- Polymorphism in Java and C++

Prof. Necula CS 164 Lecture 25

2

## Example: Lists

---

- A list of teams, each team is a list of students
  - Get the student ID of the first student in the 12<sup>th</sup> team

```
... let teams: TeamList <- ... in teams.nth(12).nth(1).sid() ...
class TeamList {
  data: StudentList;
  next: TeamList; next(): TeamList { next };
  init(d: StudentList, n: TeamList) { data <- d; next <- n }
  nth(i: Int): StudentList { if i = 1 then data else next.nth(i - 1) }
}
class StudentList {
  data: Student;
  next: StudentList; next(): StudentList { next };
  init(d: Student, n: StudentList) { data <- d; next <- n }
  nth(i: Int): Student { if i = 1 then data else next.nth(i - 1) }
}
class Student { sid(): Int { ... } }
```

Prof. Necula CS 164 Lecture 25

3

## Example: Notes

---

- Code for team lists and student lists is identical
  - Except for the typing annotations

```
class List {
  data: ?;
  next: List; next(): List { next };
  init(d: ?, n: List) { data <- d; next <- n }
  nth(i: Int): ? { if i = 1 then data else next.nth(i - 1) }
}
```

- Type checker forces us to write/maintain several copies of the same code
  - Bad software engineering practice

Prof. Necula CS 164 Lecture 25

4

## Workaround: Object to the Rescue

---

- We can write a generic list of `Object`

```
class ObjectList {
  data: Object;
  next: List; next() : List { next };
  init(d: Object, n: List) { data <- d; next <- n }
  nth(i: Int) : Object { if i = 1 then data else next.nth(i - 1) }
}
```
- We can create lists of students and of teams

```
... let empty : ObjectList in
  let students: ObjectList <- (new ObjectList).init(new Student, empty) in
  let teams: ObjectList <- (new ObjectList).init(students, empty) in ...
```
- Both calls to `init` are well-typed because `Student` and `ObjectList` are subtypes of `Object`

Prof. Necula CS 164 Lecture 25

5

## Difficulties with Object

---

- It is harder to extract an element from a list
- Instead of

```
... let teams: TeamList <- ... in teams.nth(12).nth(1).sid() ...
```
- Write

```
... let teams: ObjectList <- ... in
  case teams.nth(12) of
    slist: ObjectList => case slist.nth(1) of
      s : Student => s.sid() esac;
  esac
```
- This code is
  - Ugly, slow, might have runtime errors

Prof. Necula CS 164 Lecture 25

6

## Difficulties with Object (II)

---

- We can have lists of anything and everything !
  - The same list can have more than one kind of element  
`(new ObjectList).init(1, (new ObjectList).init(true, empty))`
- Most often we want homogeneous collections
- The type checker cannot help us keep lists homogeneous
  - Such errors would be manifested as "case" run-time errors
  
- This is the state-of-the-art in Java version 1.4
  - Will change in Java 1.5

Prof. Neula CS 164 Lecture 25

7

## Polymorphism

---

- The code for `ObjectList.nth` works for different types of lists: lists of students, integers, ...
- Code usable for several types of data is polymorphic
  - From Greek: "many shapes"
- Code usable for a single type of data is monomorphic
  - Example: `not` in Cool works only for Bool

Prof. Neula CS 164 Lecture 25

8

## Kinds of Polymorphism

---

- Subtype polymorphism: code works for all types that are subtype of a given type
  - Code written for `Student` works for `EngineeringStudent`
  - This kind of polymorphism exists in Java, C++, and Cool
- Ad-hoc polymorphism: code works for different types but uses different algorithms at different types
  - `sizeof` operator in C and C++
  - Equality comparison in Cool (pointer equality for non-basic objects, and content equality for the basic objects)
  - Related to overloading: use of the same operator or method name for a set of implementations, to be determined based on the argument types (example: `+` in most languages)

## Kinds of Polymorphism (II)

---

- Parametric polymorphism: the same code works for all types
  - Lists, stacks, collections
- In Java and Cool parametric polymorphism can be achieved using subtype polymorphism
  - Since all types are subtype of `Object`
  - But this is not the best way to achieve parametric polymorphism
- In this lecture, an elegant way to achieve parametric polymorphism

## Overview

---

- ✓ Why polymorphism ?
- Adding polymorphism to Cool
- Type checking
- Code generation
- Polymorphism in Java and C++

Prof. Neula CS 164 Lecture 25

11

## Adding Polymorphism to Cool

---

- Recall the code for lists
- We add notation to express the above code
  - Introduce a name for the type ?

```
class List {
  data: ?;
  next: List; next() : List { next };
  init(d: ?, n: List) { data <- d; next <- n }
  nth(i: Int): ? { if i = 1 then data else next.nth(i - 1) }
}
```

```
class List <t> {
  data: t;
  next: List <t>; next() : List <t> { next };
  init(d: t, n: List <t>){ data <- d; next <- n }
  nth(i: Int): t { if i = 1 then data else next.nth(i - 1) }
}
```

Prof. Neula CS 164 Lecture 25

12

## Adding Polymorphism to Cool (II)

---

- We need a notation for using a polymorphic class

```
let students : List <Student> <- new List <Student> in
let teams: List < List< Student>> <- (new List <List<Student>>).init(students,
                                                                    empty) in
students.nth(1).sid() ...
```
- If `List<t>` is a generic list class then `List<T>` is a list of `T`
- `t` is a type variable
  - Stands for a real type that will be specified when the class is instantiated
  - The scope of `t` is the entire class definition

Prof. Neula CS 164 Lecture 25

13

## Adding Polymorphism to Cool (III)

---

- We allow the definition of monomorphic and polymorphic classes
  - We use letters `C`, `D` for monomorphic and `P`, `R` for polymorphic
- New language of types:

```
T ::= C | P<T> | t
```

  - Polymorphic classes have only one parameter
- There are restrictions where `t` can appear as a type
  - Not allowed: "`new t`", "`case e of x : t => ...`", dispatch on something of type `t` (to simplify the implementation)
  - `t` can appear only in the body of a class "`class P<t> { ... }`"

Prof. Neula CS 164 Lecture 25

14

## Adding Polymorphism to Cool (IV)

---

- We can inherit from a polymorphic class

```
class ReplList<t> inherits List<t> { setcar(x: t) : Object { ... } }
class Team inherits List<Students> { nickname(): String { ... } }
```
- We can use a `ReplList<T>` where a `List<T>` is expected
  - We say that that `ReplList<T> ≤ List<T>`, for any type `T`

## Overview

---

- ✓ Why polymorphism ?
- ✓ Adding polymorphism to Cool
- Type checking
- Code generation
- Comparison with *Generic Java* and *C++ templates*

## Type Checking for Polymorphism

---

- Typing judgment before:  $O, M, C \vdash e : T$
- We need to reject the programs
  - `class ReplList<t> { setcar(x: s) : Object { ... } }`, or
  - `class ReplList<t> { setcar(x: List<s>) : Object { ... } }`, or
  - `class ReplList { setcar(x: t) : Object { ... } }`
- Must keep track of the type variable in scope (if any)
- Typing judgment now:  $O, M, T_0 \vdash e : T$ 
  - $T_0$  is the class whose definition is being checked:  $C$  or  $P<t>$
  - This way  $T_0$  keeps track of the name of the type variable

Prof. Necula CS 164 Lecture 25

17

## Type Validity

---

- We add a new judgment to check types

$$T_0 \vdash T : \text{type}$$

- $T$  is a valid type in the class definition  $T_0$

$$\frac{}{T_0 \vdash C : \text{type}} \quad \frac{}{P<t> \vdash t : \text{type}} \quad \frac{T_0 \vdash T : \text{type}}{T_0 \vdash P<T> : \text{type}}$$

- We must check types wherever they can occur
  - Example: modified typing rule for `let`

$$\frac{T_0 \vdash T : \text{type} \quad O[T_1/x], M, T_0 \vdash E : T_1}{O, M, T_0 \vdash \text{let } x : T \text{ in } E : T_1}$$

Prof. Necula CS 164 Lecture 25

18

## Polymorphic Method Invocation

---

- Recall the code

```
class List <t> {
  data: t;
  next: List <t>; next(): List <t> { next };
  init(d: t, n: List <t>){ data <- d; next <- n }
  nth(i: Int): t { if i = 1 then data else next.nth(i - 1) }
}
... let teams : List<List<Student>> <- ... in teams.nth(12).nth(1).sid ...
```

- Must refine checking of method invocation:
  - check that  $O, M, T_0 \vdash \text{teams.nth}(12) : \text{List}\langle \text{Student} \rangle$
  - check that  $O, M, T_0 \vdash \text{teams.nth}(12).\text{nth}(1) : \text{Student}$

## Polymorphic Method Invocation (II)

---

- Recall the typing rule for method invocation (one arg.)

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : C_0 \\ O, M, C \vdash e_1 : C_1 \\ M(C, f) = (C'_1, C'_2) \\ C_1 \leq C'_1 \end{array}}{O, M \vdash e_0.f(e_1) : C'_2}$$

- where  $M(C, f) = (C'_1, C'_2)$  if class  $C$  has a method declared as " $f(x: C'_1) : C'_2 \{ \dots \}$ "
- We define  $M(T_0, f) = (T'_1, T'_2)$  in a similar way
  - Except that if  $T_0 = P\langle t \rangle$  then  $t$  may appear in  $T'_1$  and  $T'_2$

## Polymorphic Method Invocation (III)

---

- Recall the code

```
class List <t> {  
  ...  
  init(d: t, n: List <t>) { data <- d; next <- n }  
  nth(i: Int): t { if i = 1 then data else next.nth(i - 1) }  
}  
... let teams : List<List<Student>> <- ... in teams.nth(12).nth(1).sid ...
```

- Then
  - $M(\text{List}\langle t \rangle, \text{init}) = (t, \text{List}\langle t \rangle)$
  - $M(\text{List}\langle t \rangle, \text{nth}) = (\text{Int}, t)$
- In the call `teams.nth(12)` the type variable  $t$  stands for `List<Student>`
  - In each call the type variable can be instantiated to a different type

Prof. Necula CS 164 Lecture 25

21

## Polymorphic Method Invocation (IV)

---

- An additional typing rule for method invocation

$$\frac{\begin{array}{l} O, M, T_0 \vdash e_0 : P\langle T \rangle \\ O, M, T_0 \vdash e_1 : T_1 \\ M(P\langle t \rangle, f) = (T_1', T_2') \\ T_1 \leq T_1'' \end{array}}{O, M \vdash e_0.f(e_1) : T_2''}$$

- where  $T_1''$  is the result of substituting  $t$  with  $T$  in  $T_1'$
- and  $T_2''$  is the result of substituting  $t$  with  $T$  in  $T_2'$
- There is also the old rule for  $O, M, T_0 \vdash e_0 : C$
- We do not allow  $O, M, T_0 \vdash e_0 : t$ 
  - Also to simplify the implementation (see later)

Prof. Necula CS 164 Lecture 25

22

## Expanded Definition for the Subtyping Operator

---

- The definition of the subtyping operator is extended  
transitivity: if  $T_1 \leq T_2$  and  $T_2 \leq T_3$  then  $T_1 \leq T_3$   
 $C \leq D$  if  $C$  inherits  $D$  { ... }  
 $C \leq P\langle T \rangle$  if  $C$  inherits  $P\langle T \rangle$  { ... }  
 $P\langle T \rangle \leq C$  if  $P\langle t \rangle$  inherits  $C$  { ... }  
 $P\langle T \rangle \leq R\langle T \rangle$  if  $P\langle t \rangle$  inherits  $R\langle t \rangle$  { ... }  
 $t \leq t \leq \text{Object}$
- The rule  $t \leq \text{Object}$  is allowed because any instantiation of  $t$  will be a subtype  $\text{Object}$
- Conclusion: values of type  $t$  are only moved around (into fields and variables of type  $t$ ), or used as  $\text{Object}$

## Overview

---

- ✓ Why polymorphism ?
- ✓ Adding polymorphism to Cool
- ✓ Type checking
- Code generation
- Comparison with *Generic Java* and *C++ templates*

## Code Generation for Polymorphism

---

- Polymorphism (in Cool) adds convenience and stronger type checking but not new expressiveness
- This suggests the use of source-to-source transformation as an implementation strategy
- We want to generate a (regular) Cool class for each polymorphic class
  - Called homogeneous compilation scheme
- Observation:
  - Values of type  $\dagger$  are only moved around in the body of class  $C$
  - Must be able to instantiate  $\dagger$  to any type
- We compile  $\dagger$  as `Object` !

Prof. Necula CS 164 Lecture 25

25

## Homogeneous Compilation of Polymorphism

---

- All occurrences of  $P<...>$  are changed to  $P$
- All occurrences of  $\dagger$  are changed to `Object`

```
class List <T> {  
  data: Object  
  init(d: Object, n: List <T>) { data <- d; next <- n }  
  nth(i: Int): Object { if i = 1 then data else next.nth(i - 1) }  
}
```

- But what about method invocation
  - No need to worry about a method with parameter type  $\dagger$
  - Must do a "cast" for return values of methods with return type  $\dagger$

```
let s : Student <- (new List<Student>).nth(1) in ...  
becomes
```

```
let s : Student <- case (new List).nth(1) of s': Student => s' esac in ...
```

Prof. Necula CS 164 Lecture 25

26

## Advanced Topic: Type-Passing Compilation

---

- Luckily we do not need to generate code for "new t"
  - new Object would not be correct
- If we want to allow "new t" then we must pass to all methods the prototype object of the current instantiation of t

class Alloc<t> { alloc() : t { new t } } ... let s : Student <- (new Alloc<Student>).alloc()  
becomes

class Alloc { alloc(proto: Object) : Object { proto.copy() } } ...  
... let s : Student <- case (new Alloc).alloc(Student\_protObj) of s : Student => s esac

- We can use the prototype object to implement also "case ... of x : t => ..."

## Overview

---

- ✓ Why polymorphism ?
- ✓ Adding polymorphism to Cool
- ✓ Type checking
- ✓ Code generation
- Comparison with Generic Java and C++ templates

## Polymorphism in Java

---

- Original Java has only subtype polymorphism
- Starting with version 1.5 it also has parametric polymorphism
  - Based on the *Generic Java* research project
- Same syntax as in this lecture
- Similar source-to-source implementation scheme
  - With similar limitations (e.g., no "new +")

## Polymorphism in C++

---

- C++ has templates

```
template List<class T, typename R, int size> {  
    R array[size]; // Field of variable size  
    R foo = (new T).meth(sizeof( R));  
    ...  
}
```
- C++ templates are more powerful than our version of polymorphism
  - Parameters can be class names, type names, data values, or other templates
  - Can use type variables anywhere a type is expected
  - C++ templates must use a different implementation strategy

## Compilation of C++ templates

---

- Template definitions are parsed but not type checked until they are used !
- For each use, the compiler generates C++ code obtained by textual substitution:

List<MyClass, int, 20> is instantiated as: 

```
class List_MyClass_int_20 {
    int array[20]; // Field of variable size
    int foo = (new MyClass).meth(sizeof( int));
}
```

List<Foo, double, 10> is instantiated as: 

```
class List_Foo_double_10 {
    double array[10]; // Field of variable size
    double foo = (new Foo).meth(sizeof(double));
}
```

- Instantiations are compiled and type checked independently !

Prof. Neula CS 164 Lecture 25

31

## Compilation of C++ Templates

---

- This compilation scheme is called heterogeneous
- C++ compiler must avoid generating multiple identical template instantiations
  - Even if they occur in different files of a project !
- If you are not careful, a significant code explosion
  - But you only need to maintain one copy

```
#include <iostream>
void main() { cout << "Hello world!"; }
```
  - Generates 330,000 lines of code once it instantiates the necessary templates from STL !
  - Luckily machines are fast nowadays
- Templates are one of the hardest-to-get-right issues in a C++ compiler

Prof. Neula CS 164 Lecture 25

32

## Conclusion

---

- Several forms of polymorphism
- Parametric polymorphism adds stronger type checking
  - At the cost of additional type-checking machinery
- Parametric polymorphism is a feature of advanced languages
- Java (version 1.5) and C++ have param. polymorphism
  - Very different compilation schemes though