

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 12th 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
- Type checker forces us to write/maintain several copies of the same code
 - Bad software engineering practice

```
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) }
}
```

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. Necla 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. Necla 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)

Prof. Necla CS 164 Lecture 25

9

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

Prof. Necla CS 164 Lecture 25

10

Overview

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

Prof. Necla CS 164 Lecture 25

11

Adding Polymorphism to Cool

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

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

Prof. Neucula 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  $ReplList<T> \leq List<T>$ , for any type  $T$

Prof. Neucula CS 164 Lecture 25

15

### Overview

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

Prof. Neucula CS 164 Lecture 25

16

### 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. Neucula 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  $T_0$
$$\frac{T_0 \vdash C : \text{type}}{T_0 \vdash C : \text{type}} \quad \frac{P<t> \vdash t : \text{type}}{T_0 \vdash P<t> : \text{type}} \quad \frac{T_0 \vdash T : \text{type}}{T_0 \vdash P<t> : \text{type}}$$
- 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. Neucula 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  $O, M, T_0 \vdash \text{teams.nth}(12) : \text{List}<\text{Student}>$
  - check that  $O, M, T_0 \vdash \text{teams.nth}(12).\text{nth}(1) : \text{Student}$

Prof. Neucula CS 164 Lecture 25

19

### Polymorphic Method Invocation (II)

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

$$\frac{O, M, C \vdash e_0 : C_0 \quad O, M, C \vdash e_1 : C_1 \quad M(C_0, f) = (C_1', C_2') \quad C_1 \leq C_1'}{O, M \vdash e_0.f(e_1) : C_2'}$$

- 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  $T_0 = P<T>$  then  $\dagger$  may appear in  $T_1'$  and  $T_2'$

Prof. Neucula CS 164 Lecture 25

20

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

Prof. Neucula CS 164 Lecture 25

21

### Polymorphic Method Invocation (IV)

- An additional typing rule for method invocation

$$\frac{O, M, T_0 \vdash e_0 : P<T> \quad O, M, T_0 \vdash e_1 : T_1 \quad M(P<T>, f) = (T_1', T_2') \quad T_1 \leq T_1'}{O, M \vdash e_0.f(e_1) : T_2'}$$

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

Prof. Neucula CS 164 Lecture 25

22

### Expanded Definition for the Subtyping Operator

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

Prof. Neucula CS 164 Lecture 25

23

### Overview

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

Prof. Neucula CS 164 Lecture 25

24

## 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  $t$  are only moved around in the body of class  $C$
  - Must be able to instantiate  $t$  to any type
- We compile  $t$  as **Object**!

Prof. Necula CS 164 Lecture 25

25

## Homogeneous Compilation of Polymorphism

- All occurrences of  $P<...>$  are changed to  $P$
  - All occurrences of  $t$  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 t
 - Must do a "cast" for return values of methods with return type t
- ```
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 $x : t \Rightarrow \dots$ "

Prof. Necula CS 164 Lecture 25

27

Overview

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

Prof. Necula CS 164 Lecture 25

28

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 t ")

Prof. Necula CS 164 Lecture 25

29

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

Prof. Necula CS 164 Lecture 25

30

## 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. Neucula 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. Neucula 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

Prof. Neucula CS 164 Lecture 25

33