Classes and Inheritance

1

Inheritance

- Inheritance means : Using some definitions of a baseclass in a subclass without having to write them again.
- Inheritance ≠ Subtyping !
- Inheritance is about code resuse, subtyping about substitutability.
- Question : Find an example for subtyping where inheritance is not used.
- Question : Find an example for inheritance where subtyping is not used.

2

Modelling Inheritance in Funnel

Inheritance is supported indirectly through the with construct.

With imports all fields of a given record into the current environment.

If r : $\{x_1 : T_1,..., x_n : T_n\}$ then with r is equivalent to :

def $x_1 = r.x_1,..., x_n = r.x_n$

This leads to the typing rule :

(With) $$\frac{\Gamma \vdash E : \{x_1 : T_1,..., x_n : T_n\} \quad \Gamma, x_1 : T_1,..., x_n : T_n \vdash F : U}{\Gamma \vdash \textbf{with } E; F : U}$$

3

Classes

- Many object-oriented languages are organized around classes.
- A class fulfils two roles :
  - It defines a *type* of objects with a common interface.
  - It defines a way to *create* objects of the class.

Example : Assume the following class definition in a (as yet hypothetical) object-oriented extension of Sunnel.

```
class Point (x: Int) = {
        private var curpos :=x
        def position: Int = curpos
        def move (delta: Int) = curpos := curpos + delta
}
```

(Note : Instead of a Point constructor, as in Java, we can add value parameters to the class itself).

4

Classes (2)

- This can be expanded to a type definition :

```
type Point = {
    def position: Int
    def move (delta: Int): ()
}
```

- and a value definition :

```
val Point = {
    def new (x: Int): Point = {
        var curpos :=x
        def position: Int = curpos
        def move (delta: Int) = curpos := curpos + delta
    }
}
```

- Now, points can be created by Point.new (pos).
- Note that private in the OOP language is expressed by projecting to a supertype without the field here. (how accurate is this?).

5

Static Class Members

- Static class members go in the value part of a class.
- Example :

```
class Point (x: Int) = {
    …
static
    def origin = 0
}
```

- This leads to the value definition :

```
val Point = {
    def origin = 0
    def new (x: Int): Point = …
}
```

6

Inheritance

- Let 's now define a subclass which inherits from Point.

```
class ColoredPoint (x: Int, c:Color) extends Point (x) = {
    private var curc := c
    def color = curc
    def move (delta: Int) = { super.move (delta); curc := Red }
}
```

- Here is its translation :

```
type ColoredPoint ={
    with Point
    def color: Color
    def move (delta: Int): ()
}
```

7

Inheritance (2)

```
val ColoredPoint : {
    def new (x: Int, c: Color): ColoredPoint = {
        val super = Point.new (x)
        with super
        var curc := c
        def color = curc
        def move (delta: Int) = { super.move (delta); curc := Red }
    }
}
```

Notes :

- Inheritance expressed by with.
  super is an explicit object.
  Calls to inherited methods are forwarded.

8

A Problem

- Let 's add another method

```
class Point (x: Int) = {
    private var curpos := x
    def position: Int = curpos
    def move (delta: Int) = curpos := curpos + delta
    def reflect = move (-2 * position)
}
```

- What happens when we reflect a ColoredPoint ?

9

Dynamic Binding

- Its reasonable for Point.reflect to call the implementation of move which is current for the actual object. Hence, if we reflect a ColoredPoint, it 's color should turn Red.
- To achieve this, we introduce a special identifier this which stands for the current object :

```
class Point (x: Int) = {
    private var curpos := x
    def position: Int = curpos
    def move (delta: Int) = curpos := curpos + delta
    def reflect = this.move (-2 * position)
}
```

- How is this translated?

10

First Attempt

- Let 's take a clue from a common implementation strategy for objects (e.g. in Java) :
- There, this is passed as an addition parameter to each method.
- Example :

```
val Point = {
    def new (x: Int): Point = {
        var curpos := x
        def position (this: Point): Int = curpos
        def move (this: Point, delta: Int) = curpos := curpos + delta
        def reflect (this: Point) = this.move (this, -2 * position)
    }
}
```

11

First Attempt (2)

```
val ColoredPoint = {
    def new (x: Int, c: Color): ColoredPoint = {
        val super = Point.new (x)
        with super
        var curc := c
        def color (this: ColoredPoint) = curc
        def move (this: ColoredPoint, delta: Int) = {
            super.move (this, delta) ; curc := Red
        }
    }
}
```

- Problem: ColoredPoint.move no longer overrides Point.move, because the types don 't match. Compare with type rule (ARROW).
  Also: this is unavailable for in an object initializer (since it is not represented as a method).

12

Second Attempt

- Pass the current value of this as parameter to the object creator method (now called newSuper):

```
val Point = {
    def newSuper (this: Point, x: Int): Point = {
        var curpos := x
        def position = curpos
        def  move (delta: Int) = curpos := curpos + delta
        def reflect = this.move (-2 * position)
    }
}
```

13

Second Attempt (2)

```
val ColoredPoint = {
  def newSuper (this: ColoredPoint, x: Int, c: Color): ColoredPoint = {
      val super = Point.newSuper (this, x)
      with super
      var curc := c
      def color = curc
      def move (delta: Int) = {
          super.move (delta) ; curc := Red
      }
  }
}
```

14

Object-Creation

- Object-creation invokes newSuper.
- But newSuper requires the created object to be passed as a parameter!
- How can this be made to work ?
- E.g. Point.new (x) is the (least-defined) object P which satisfies :

  P = Point.newSuper (P, x)
- How can this object be defined ?

15

Attempts

```
val P = Point.newSuper (P, x)
```

does not work, because val does not admit recursive definitions.

```
def P = Point.newSuper (P, x)
```

does not work, because every occurrence of P would create a new object.

16

Recursive Bindings

- We need a way to define non-functional values which refer to themselves.
- Introduce a new construct for this :

  let *x : T = E ; F*
- Typing rule :

$$\text{(LET)}\quad \frac{\Gamma, x : T \vdash E : T \qquad \Gamma, x : T \vdash F : U}{\Gamma \vdash \text{let } x : T = E ; F : U}$$

17

Denotational Semantics of Recursive Bindings

- It 's natural to model recursion with fixedpoints.
- So here we go : First, define a transformation function.

  def cycleF (cycle) = { **val** nx = cycle ; **def** next = nx }
- Next, define cycle to be the least fixed point of cycleF, i.e. the limit of the chain :

  ⊥, cycleF (⊥), cycleF (cycleF(⊥)), …
- What do we get ?

  ⊥ , ⊥ , ⊥, ...

18

Meaning of Recursive Bindings

- The semantics of recursive bindings is a bit subtle.
- When evaluating a recursive binding, it's OK to reuse the defined name on the right-hand side :

```
let cycle = { val nx = cycle ; def next = nx }   / / OK
```

- But we cannont select the defined name while evaluating the binding:

```
let badcycle = { val nx = cycle.next ; def next = nx } / / Run-time error
```

- This looks a bit like lazy evaluation!

19

A Refinement

- The previous example has shown that the start value of the fixed point limit was « too small ».
- This can be corrected by starting the chain with a larger initial value.
- Instead of ⊥, use a record with fields which are all ⊥.
- Then the chain is :

```
{next: ⊥}, cycleF {next: ⊥}, cycleF (cycleF {next: ⊥})
```

- which evaluates to:

```
{next= ⊥}, {next= {next= ⊥}}, {next= {next= {next= ⊥}}}
```

- The limit of this chain is the record which refers to itself via its next field (that 's what we want).

20

Lazy Evaluation

- We have given a semantics of lazy evaluation of let-bound values in a strict language.
- Normally, lazy evaluation is described as a property of the *functions* which get applied to values.
- A lazy function is one which does not necessarily map a $\perp$ argument to a $\perp$ result.
- But our functions are strict; they do map $\perp$ to $\perp$!
- Instead, we define lazy evaluation as a property of the value itself.

21

Implementation

- The previous discussion gave a denotational semantics of the purely functional case.
- We still have to explain how to implement the limit of the chain.
- We also have to take side effects and concurrency into account.
- This will be done by giving an implementation of let as a functional net.

22

Implementation (2)

- Let $T = \{ f_1 : T_1, \ldots, f_n : T_n \}$.
- Then **let** x : T = E translates to :

```
val x = {
    def   x.f1 & undef        =   x.f1 & defined(E),
          x.f1 & defined(y)   =   y.f1 & defined(y),
    . . .
          x.fn & undef        =   x.fn & defined(E),
          x.fn & defined(y)   =   y.fn & defined(y);
    x & undef
}
```

23

Implementation (3)

- This definition defines the outer *x* as a record value.
- When selecting a field of the record, we test whether the record is still undefined.
- If it is still undefined, we define the record to be the result of *E* and select again.
- If it is already defined with result *y*, we select the same field in *y*.
- Question : What happens if *E* refers to *x*?
  What happens if *E* selects a field in *x*?

24

A Variant

- For the purposes of recursive bindings, lazy evaluation is overkill.
- It would be sufficient to delay evaluation until the end of the recursive definition, and not wait until the time the defined identifier is first used.
- Invent a new binding for this :

  valrec *x : T = E ; F*

- How can valrec be implemented ? (Hint : take the implementation of let and try to simplify.

25

Object Creation

- Object Creation can now be expressid through a let or a valrec:

  valrec P = Point.newSuper (P,x)

- We can then add the following new methods to Point and ColoredPoint

```
val Point = {
    def newSuper …
    def new (x: Int): Point = {
        valrec this = newSuper (this, x) ; this
    }
}
val ColoredPoint = {
    def newSuper …
    def new (x: Int, c: Color): ColoredPoint = {
        valrec this = newSuper (this, x, c)) ; this
    }
}
```

26

Refinements

- Question: In Java a class marked final cannot be subclassed. Can this be modelled in Funnel?
- Question: In Java, classes and methods may be declared as abstract.
  - An abstract method does not have a body, needs to be overridden in subclasses.
  - A class is abstract if it has abstract methods.
  - Instances of abstract classes cannot be created (but instances of non-abstract subclasses can).
- How can this be modelled ?

27

Abstract Classes

- Look at the signatures of the newSuper methods of Point and ColoredPoint:

```
val Point = {
    def newSuper (this: Point, x: Int): Point = …
}
val ColoredPoint = {
    def newSuper (this: ColoredPoint, x: Int, c: Color): ColoredPoint = …
}
```

- Note that the type of the this parameter always equals the return type of the function.
- This changes in the presence of abstract methods.
- Abstract methods can 't be part of the result type of newSuper (since we don 't have a body for them).
- But they need to be part of the this parameter record (since they may be referred to).

28

Example

- Consider an abstract superclass of Point:

```
class Movable = {
    abstract def position: int
    abstract def move (delta: Int)
    def reflect = this.move ( - position * 2)
}
```

- Translating this yields the types:

```
type ConcreteMovable = {
    def reflect: ()
}
type Movable = {
    def position: int
    def move (delta: Int): ()
    def reflect: ()
}
```

29

Example (2)

- The Movable type contains all methods, abstract or not.
- The ConcreteMovable type contains all non-abstract methods.
- The Movable value is as follows :

```
val Movable = {
    def newSuper (this:Movable): ConcreteMovable = {
        def reflect = this.move ( - position * 2)
    }
}
```

- Note that new is missing, since we can 't create instances of Movable.
- New could not be defined anyway, since

```
valrec this = Movable.newSuper (this)
```

gives a type error.

30

Inheriting Abstract Classes

- Let 's now make Point inherit from Movable.

```
class Point (x: Int) extends Movable = {
    private var curpos := x
    def position: Int = curpos
    def move (delta: Int) = curpos := curpos + delta
}
```

- Here, the definition of reflect is inherited from **Movable**.
- The value translation of Point is :

31

Inheriting Abstract Classes (2)

```
val Point = {
    def newSuper (this: Point, x: Int): Point = {
        val super = Movable.newSuper (this)
        with super
        var curpos := x
        def position = curpos
        def move (delta: Int) = curpos := curpos + delta
    }
    def  new (x: Int): Point = {
        valrec this = newSuper (this, x) ; this
    }
}
```

- Question: In C++, a protected field of an object can be accessed only from the object itself (where the access is found either in the same class or in a subclass. How can this be modelled ?

32

Summary

- We have seen how key constructs of object-oriented programming can be represented and formalized.
  - A class defines a record type and a record value.
  - The value contains static fields and two creation functions :
    - new is used to create an object of the class from the outside.
    - newSuper is used to create superclass instance for an object of the subclass.
  - Inheritance is modelled by including the superclass object with a with.
  - The superclass object know about the identity of the subclass object through the this parameter which is passed to newSuper.
  - This scheme is called *delegation.*
  - The record type defines the public interface of objects of the class.
  - Extending a record type *T* creates a subtype, which is compatible with *T*.

33