Objekt-orientiertes Programmieren

5 wesentliche Aspekte

1. Polymorphie
   - Ein Interface kann mehrere Implementationen besitzen.
   - Dynamic dispatch: Object entscheidet, welcher Code ausgeführt wird.
2. Kapselung
   - Nur Methoden des Objektes können auf Instanzenvariablen direkt zugreifen.
3. Subtyping
   - Ein Objekt mit mehr Funktionalität (Methoden) kann an jeder Stelle verwendet werden, an der ein Objekt mit weniger Funktionalität erwartet wird.
4. Vererbung
   - Verhalten und Implementation kann an Subklasse weitergegeben werden.
5. Späte Bindung (self, this)
   - Vituelle Methoden.

Im folgenden zeigen wir am Beispiel eines Zählers (counter), wie wir die obigen Konzepte in unserer funktionalen Sprache simulieren können.

I Objekte, Methoden

Interface.

```
Counter = { get: Uni -> Int , inc: Unit -> Unit }
```

Objekt.

```
c = let x = ref 0 in
        { get = \lambda _:Unit. !x,
          inc = \lambda _:Unit. x := !x + 1 }
c : Counter
```

Methodenaufruf.

```
c.get(), c.inc()
inc3 = \lambda c: Counter. (c.inc(); c.inc(); c.inc())
```

II Konstruktor

Objektkonstruktoren bezeichnen wir mit `new...`

```
newCounter : Unit -> Counter
newCounter = \lambda _:Unit. let x = ref 0 in
        { get = \lambda _:Unit. !x,
          inc = \lambda _:Unit. x := !x + 1 }
```

III Subtyping

```
ResetCounter={get...; inc..., reset:Unit -> Unit}
newResetCounter = \lambda _:Unit. let x = ref 0 in
        { get   = \lambda _:Unit. !x,
          inc   = \lambda _:Unit. x := !x + 1,
          reset = \lambda _:Unit. x := 0 }
```

IV Bündelung der Instanzen-Variablen

```
CounterRep = { x: Ref Int }
newCounter = \lambda _:Unit.
    let r = {x=ref 0 } in
        { get = \lambda _:Unit. !(r.x),
          inc = \lambda _:Unit. r.x := !(r.x) + 1 }
```

V Klassen

0.1 Definition. Interface
Ein Datensatz-Typ, in dem jeder Eintrag ein Funktionstyp ist.

0.2 Definition. Objekt
Ein Term vom Typ "Interface".

0.3 Definition. Methode
Ein Bestandteil eines Objektes.

0.4 Definition. Klasse
Die Sammlung der Objekte mit identischer Implementation der Methoden.

```
CounterRep = { x: Ref Int}
counterClass: CounterRep -> Counter
counterClass
     = \lambda r: CounterRep.
        {get = \lambda _ : Unit. !(r.x),
         inc = \lambda _ : Unit. r.x := !(r.x) + 1 }
```

Vererbung

```
ResetCounter = { get..., inc..., reset: Unit ->Unit }

resetCounterClass : CounterRep -> ResetCounter
resetCounterClass = \lambda r: CounterRep.
     let super = counterClass r in
         { get = super.get,
           inc = super.inc,
           reset = \lambda _: Unit. r.x := 0 }

newResetCounter = \lambda _: Unit. let r = {x = ref 0} in resetCounterClass r
```

VI Zusätzliche Instanzenvariablen

```
BackupCounterRep = { x: Ref Int, b: Ref Int}
BackupCounter = { get..., inc..., reset..., backup: Unit -> Unit }

backupCounterClass : BackupCounterRep -> BackupCounter
backupCounterClass = \lambda r : BackupCounterRep.
     let super = counterClass r in
         { get = super.get,
           inc = super.inc,
           reset = \lambda _: Unit. r.x := !(r,b),
           backup = \lambda _: Unit. r.b := !(r.x) }
```

VII Self (ohne dynamische Bindung)

Ziel: Eine Methode kann andere Methoden des gleichen Objektes aufrufen. (Rekursion)

```
SetCounter = { get : Unit -> Unit,
               set : Int -> Unit,
               inc : Unit -> Unit }

setCounterClass : CounterRep -> SetCounter
setCounterClass = \lambda r : CounterRep.
      fix self : SetCounter.
      { get = \lambda _ : Unit. !(r.x),
        set = \lambda i : Int. r.x := i,
        inc = \lambda _ : Unit.
              self.set (self.get()+1) }

newSetCounter : Unit -> SetCounter
newSetCounter = \lambda _:Unit
      let r = {x = ref 0 } in setCounterClass r
```

VIII Self (mit dynamischer Bindung)

```
SetCounter = { get : Unit -> Unit,
               set : Int -> Unit,
               inc : Unit -> Unit }

setCounterClass : CounterRep -> SetCounter -> SetCounter
setCounterClass = \lambda r : CounterRep.
      \lambda self : SetCounter.
      { get = \lambda _ : Unit. !(r.x),
        set = \lambda i : Int. r.x := i,
        inc = \lambda _ : Unit.
              self.set (self.get()+1) }

newSetCounter : Unit -> SetCounter
newSetCounter = \lambda _:Unit
      let r = {x = ref 0 } in
        fix self : SetCounter. setCounterClass r self
```

Problem: Mit cbv-Auswertung hängt sich der Rechner beim Instantiieren der Klasse mit `newSetCounter` auf:

```
fix self : SetCounter. setCounterClass r self
--> setCounterClass r (fix self... setCounterClass r self)
--> setCounterClass r (setCounterClass r (fix self... setCounterClass r self ))
--> ...
```

Abhilfe

1. Man verwendet eine “lazy” Sprache, die Funktionsargumente nur bei Bedarf auswertet.

2. Man kodiert verzögerte Auswertung per Hand durch “dummy”-Abstraktionen `\lambda _:Unit` und -Applikationen `()`.

   ```
   self : Unit -> SetCounter
   self () : SetCounter

   setCounterClass : CounterRep -> (Unit -> SetCounter) -> SetCounter
   setCounterClass = \lambda r : CounterRep.
         \lambda self : Unit -> SetCounter.
         { get = \lambda _ : Unit. !(r.x),
           set = \lambda i : Int. r.x := i,
           inc = \lambda _ : Unit. self().set (self().get() + 1) }
   ```

```
newSetCounter = \lambda _:Unit
      let r = {x = ref 0 } in
        fix self : Unit -> SetCounter.
          \lambda _ : Unit. setCounterClass r self
```

Beispiel: “instrumented Counter”; zählt die Anzahl der Schreibzugriffe mit `set`. Das in der Oberklasse `setCounterClass` mit später Bindung implementierte `inc` kann unverändert übernommen werden, obwohl `set` überschrieben wird.

Achtung! Diese Implementation ist in einer strikten cbv-Sprache inkorrekt und muss nach dem Vorbild von `setCounterClass` umgearbeitet werden.

```
InstrCounterRep = { x : RefInt, a: RefInt }
InstrCounter = { get..., set..., inc..., accesses: Unit -> Int }

instrCounterClass = InstrCounterRep -> InstrCounter -> InstrCounter
instrCounterClass = \lambda r : InstrCounterRep \lambda self : InstrCounter.
      let super = setCounterClass r self
      in { get = super.get ,
           set = \lambda i : Int. ( super.set(i) ; r.a := !(r.a)+1),
           inc = super.inc ,
           accesses = \lambda _ : Unit. !(r.a) }
```