Part VII: Types for Object-Oriented Programming

Object-oriented programming poses new challenges to type systems.

- Record types
- Subtyping
- Inheritance
- Classes
- Meaning of `this`.

Example Language: Mini-Funnel

Syntax of Mini-Funnel

Name	x, y, z		
Tyvar	a, b, c		
Ident	I, J	$=$	$x \mid I.x$
Term	E, F	$=$	$x \mid E.x \mid E[\overline{T}] \mid E\ F \mid \mathbf{def}\ D\ ; E$
		$\mid$	$E\ \&\ F \mid \mathbf{nil}$
Definition	D	$=$	$\epsilon \mid D, D \mid L = E$
Left-hand side	L	$=$	$I[\overline{a}](x : T) : U \mid L\ \&\ L'$
Type	S, T	$=$	$a \mid \{x_1 : T_1, \ldots, x_n : T_n\} \mid T \to U$
		$\mid$	$\overline{a}.T \mid \mu a.T$
Lifted Type	U	$=$	$T \mid \mathbf{nil}$
Environment	Γ, Σ, Δ	$=$	$\epsilon \mid \Gamma, a \mid \Gamma, x : T$

Remarks

- We use a minimal language subset, which exemplifies most of the important typing problems.
- Other constructs can be added by deriving typing rules for a construct from the construct's expansion into the minimal language subset.
- **nil** is used as a constant for the empty process and as a name for the types of processes (which no not return a result).
- Difference between types T and lifted types U: lifted types can be **nil**, normal types cannot.
- To keep the formal treatment simple, we require explicit type annotations everywhere. We expect that it's possible to use clever type inference for inserting most of these annotations automatically.

Type Equivalence

We assume the following equivalences between types.

- a-renaming: The names of $\forall$-bound type variables don't matter. A $\forall$-prefix which binds an empty list of type variables may be dropped.

$$(a)\ \frac{\bar{b} \notin \mathrm{tv}(T)}{\forall\bar{a}.T \equiv \forall\bar{b}.[\bar{b}/\bar{a}]T} \qquad (\textsc{Empty})\ \ \forall\epsilon.T \equiv T$$

- The order of fields in a record does not matter.

$$(\textsc{Permute})\ \frac{\{i_1, \ldots, i_n\} = \{1, \ldots, n\}}{\{x_1 : T_1, \ldots, x_n : T_n\} \equiv \{x_{i_1} : T_{i_1}, \ldots, x_{i_n} : T_{i_n}\}}$$

- A recursive type $\mu a.T$ is equivalent to its (one-step) unfolding $[\mu a.T/a].T$. Two recursive types are equivalent if their *infinite unfoldings* are equivalent.

Well-formedness

$\boxed{\Gamma \vdash U \text{ WF}}$ In environment Γ, type U is well-formed. That is, all of U's type variables are listed in Γ.

$$(\textsc{Tvar-wf})\ \ \Gamma, a \vdash a \text{ WF} \qquad (\textsc{Nil-wf})\ \ \Gamma \vdash \mathbf{nil} \text{ WF}$$

$$(\textsc{Rec-wf})\ \frac{\Gamma \vdash T_1 \text{ WF} \quad \ldots \quad \Gamma \vdash T_n \text{ WF}}{\Gamma \vdash \{x_1 : T_1, \ldots, x_n : T_n\} \text{ WF}} \qquad (\mu\text{-WF})\ \frac{\Gamma, a \vdash T \text{ WF}}{\Gamma \vdash \mu a.T \text{ WF}}$$

$$(\textsc{Arrow-wf})\ \frac{\Gamma \vdash T \text{ WF} \qquad \Gamma \vdash U \text{ WF}}{\Gamma \vdash T \rightarrow U \text{ WF}} \qquad \frac{\Gamma, \bar{a} \vdash T \text{ WF}}{\Gamma \vdash \forall\bar{a}.T \text{ WF}}$$

Subtyping

Subtyping is defined by a deduction system for judgements of the form

$\boxed{S \leq T'}$ Type S is a subtype of type T.

Meaning: "Whereever a term of type T is required, a value of type S can also be passed."

This is expressed in the *subsumption rule* for type assignments (repeated later).

$$(\textsc{Sub})\ \frac{\Gamma \vdash E : S \qquad S \leq T}{\Gamma \vdash E : T}$$

Subtyping is reflexive and transitive:

$$(\textsc{Refl})\ \ U \leq U \qquad (\textsc{Trans})\ \frac{T_1 \leq T_2 \qquad T_2 \leq T_3}{T_1 \leq T_3}$$

Subtyping Rules Continued

Records with "more fields" are subtypes of records with fewer fields.

$$(\textsc{Record})\ \frac{T_1 \leq T_1' \quad \ldots \quad T_m \leq T_m'}{\{x_1 : T_1, \ldots, x_m : T_m, \ldots, x_n : T_n\} \leq \{x_1 : T_1', \ldots, x_m : T_m'\}}$$

Rule for type schemes:

$$(\textsc{Forall})\ \frac{T \leq T'}{\forall\bar{a}.T \leq \forall\bar{a}.T'}$$

Two recursive types $\mu a.S$ and $\mu b.T$ are in a subtype relationship if their inifinite unfoldings are.

Note: This is non-trivial to formalize and to type-check (but it's possible).

Subtyping for Function Types

For function types, we have the following rule:

$$(\textsc{Arrow})\ \frac{T' \leq T \qquad U \leq U'}{T \rightarrow U \leq T' \rightarrow U'}$$

Note that the subtyping relationship is *reversed* in the function arguments!

Why is this rule required?

Ome (somewhat loosely) calls this rule the *contravariance rule* for function subtyping, because subtyping is reversed for function arguments.

Structural Subtyping vs Subtyping by Declaration

We have seen an instance of *structural subtyping*, where a type is a subtype of another purely because of the structure of the two types.

Many programming languages use instead *subtyping by declaration*, where the subtyping relationship is explicitly declared.

In these languages a type declaration introduces a new type, with a given set of fields and a given (set of) supertype(s).

Example: Objects with "head" field, infinite lists:

```
type Headed [a] = { head: a}
type Stream [a] extends Headed [a] = {
   head: a, tail: () → Stream [a]
}
```


Type Aliases

In Mini-Funnel, a type definition is seen just as an abbreviation which can always be replaced by its right-hand side. (That's why we don't need to have type definitions in the abstract syntax).

Example:

```
type Headed [a] = { head: a}
type Stream [a] = μs.{
   head: a, tail: () → s
}
```

Note that

- We express recursion through μ-types.
- Stream[a] is a subtype of Headed[a] even without an explicit extends clause.

Which scheme is preferable?

Type Assignment for Expressions

$\boxed{\Gamma \vdash E : U}$ In environment Γ, term E has type U (where U can be **nil**).

$$(\textsc{Taut})\ \Gamma, x : T \vdash x : T \qquad (\textsc{Sub})\ \frac{\Gamma \vdash E : S \qquad S \leq T}{\Gamma \vdash E : T}$$

$$(\textsc{Sel})\ \frac{\Gamma \vdash E : \{x : T\}}{\Gamma \vdash E.x : T} \qquad (\textsc{TApp})\ \frac{\Gamma \vdash E : \forall \overline{a}.T \qquad \Gamma \vdash \overline{S}\ \textsc{wf}}{\Gamma \vdash E[\overline{S}] : [\overline{S}/\overline{a}]T}$$

$$(\textsc{Apply})\ \frac{\Gamma \vdash E : T \rightarrow U \qquad \Gamma \vdash F : T}{\Gamma \vdash E\ F : U}$$

$$(\textsc{Fork})\ \frac{\Gamma \vdash E : U \qquad \Gamma \vdash F : \mathbf{nil}}{\Gamma \vdash E\ \&\ F : U} \qquad (\textsc{Nil})\ \Gamma \vdash \mathbf{nil} : \mathbf{nil}$$

$$(\textsc{Def})\ \frac{\Gamma, \Sigma \vdash D : \Sigma \qquad \Gamma, \Sigma \vdash E : U}{\Gamma \vdash (\mathbf{def}\ D\ ;\ E) : U}$$

Notation The , operator on environments and records is assumed to be associative and commutative.

Furthermore, the operands Γ, Σ of a composition Γ, Σ are required to have disjoint domains.

Type Assignment for Definitions

$\boxed{\Gamma \vdash D : \Sigma}$ In environment Γ, the definitions D are well-typed and generate environment Σ. Σ binds exactly the functions defined in D. Since definitions can be recursive, it is assumed that all functions in Σ are already in Γ.

$$(\textsc{Eqn})\ \frac{\Gamma \vdash L : \Sigma\ ;\ \Delta \qquad \Sigma = I : \forall\overline{a}.T \to U, \Sigma' \qquad \Gamma, \Delta \vdash E : U}{\Gamma \vdash L = E : \Sigma}$$

$$(\textsc{Empty})\ \Gamma \vdash \epsilon : \epsilon \qquad (\textsc{Concat})\ \frac{\Gamma \vdash D_1 : \Sigma_1 \qquad \Gamma \vdash D_2 : \Sigma_2}{\Gamma \vdash D_1, D_2 : \Sigma_1 \wedge \Sigma_2}$$

Notation

- $\wedge$ is deep intersection, defined on types and environments consisting only of value bindings as follows:

$$\begin{array}{llll}
S \wedge T & = & S & \text{if } S = T \\
& = & \{\Gamma \wedge \Sigma\} & \text{if } S = \{\Gamma\}, T = \{\Sigma\} \\
& & \text{is undefined otherwise} & \\
(\Gamma \wedge \Sigma)(x) & = & \Gamma(x) & \text{if } x \in \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Sigma) \\
& = & \Sigma(x) & \text{if } x \in \mathrm{dom}(\Sigma) \setminus \mathrm{dom}(\Gamma) \\
& = & \Gamma(x) \wedge \Sigma(x) & \text{if } x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Sigma)
\end{array}$$

- We sometimes write bindings of the form $I : T$, where I is a qualified identifier. The environment $I.x : T$ is taken to be equal to $I : \{x : T\}$.

Type Assignment for Left-hand Sides

$\boxed{\Gamma \vdash L : \Sigma\ ;\ \Delta}$ In environment Γ, left-hand-side L generates function environment Σ and local environment Δ.

$$(\textsc{Fun})\ \frac{\Gamma, \overline{a} \vdash T\ \textsc{wf} \qquad \Gamma, \overline{a} \vdash U\ \textsc{wf}}{\Gamma \vdash (I[\overline{a}](x : T) : U) : (I : \forall\overline{a}.T \to U)\ ;\ (\overline{a}, x : T)}$$

$$(\textsc{Join})\ \frac{\Gamma \vdash L : \Sigma\ ;\ \Delta \qquad \Gamma \vdash L' : \Sigma'\ ;\ \Delta' \qquad \Sigma' = (I : \forall\overline{a}.T \to \mathbf{nil})}{\Gamma \vdash L\ \&\ L' : \Sigma \wedge \Sigma'\ ;\ \Delta, \Delta'}$$

Derived Constructs

Add **val** definitions to Mini-Funnel:

$$\text{Term} \quad E, F \quad = \quad \ldots \mid \textbf{val}\ x = E\ ;\ E'$$

Typing rule for **val**:

Add sequencing to Mini-Funnel:

$$\text{Term} \quad E, F \quad = \quad \ldots \mid E\ ;\ E'$$

Typing rule:

Type Inference

The presented type system needs a lot of explicit type information.

Can we infer this information?

Constraints, as in the Hindley/Milner system don't work here so well:

- Because of subsumption, we get subtyping constraints $S \leq T$ rather than equality constraints $T = U$. Hence, unification is not applicable.
- Even without subtyping, $\forall$-quantifiers would make unification undeciable.

Alternative: Use *local* type inference, to infer some type annotations.

Local type inference computes types by propagating information from some part of the tree to neighboring parts.

Example: Given

```
def f(x: Int) = x + x
```

infer f's result type to be Int.

Example: Given

```
x: Int, xs: List[Float]
Int ≤ Float
cons: ∀a.(a, List[a]) → List[a]
```

infer the missing type parameter in cons (x, List[xs]) to be [Float]:

```
cons [Float] (x, List[xs])
```

Generally, infer missing type parameters to be those types which make the actual arguments match the formal parameters and which minimize the result type.

An error results if no optimal type parameters exist.

Example:

Given

```
type ListVisitor [a,b] = {
    def Nil: b
    def Cons (x: a, xs: List [a]): b
}
type List [a] = {
    def match [b] (v: ListVisitor [a, b]): b
}
```

infer that in the definition

```
def append (xs: List[String], ys: List[String]) = {
    xs.match {
        def Nil = ys
        def Cons (x, xs1) = List.Cons (x, append (xs, ys))
    }
}
```

the following holds:

- The record argument of xs.match is a ListVisitor[String,b] for some unknown b.
- Therefore, the full types of the Nil and Cons fields are:

  Nil: b
  Cons: (x: String, xs: List [String]) → b

  (Parameter types are inferred).
- Therefore, the result type of the visitor is List[String], which is then also the type parameter for match.

Fully typed output of type inference:

```
def append (xs: List[String], ys: List[String]): List[String] = {
   xs.match [List[String]] {
      def Nil: List[String] = ys
      def Cons (x: String, xs1: List[String]): List[String] =
         List.Cons [String] (x, append (xs, ys))
   }
}
```


See also:

- Benjamin Pierce and David Turner; Local Type Inference; Proc. ACM Symposium on Principles of Programming Languages, 1996. (their techniques solve the first two examples).
- Our current research (solves all three).

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.

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

$$\mathbf{def}\ x_1 = r.x_1, \ldots, x_n = r.x_n$$

This leads to the typing rule:

$$(\textsc{With})\ \frac{\Gamma \vdash E : \{x_1 : T_1, \ldots, x_n : T_n\} \qquad \Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash F : U}{\Gamma \vdash \mathbf{with}\ E\ ;\ F : U}$$

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 Funnel.

```
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).

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 supertpe without the field here. (How accurate is this?)

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 = ...
}
```


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): ()
}
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.

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?

Dynamic Binding

It's 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?

First Attempt

Let's take a clue from a common implementation strategt for objects (e.g. in Java):

There, this is passed as an additional 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)
    }
}
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).

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)
    }
}
```



```
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
        }
    }
}
```


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?

Proposal: Form a fixedpoint.

E.g. Point.new (x) is the (least-defined) object P which satisfies:

P = Point.newSuper (P, x)

How can this object be defined?

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.

```
var P := null
P := Point.newSuper (P, x)
```

does not work, since the this parameter passed to Point.newSuper is null.

Recursive Bindings

We need a way to define non-functional values which refer to themselves.

Introduce a new construct for this:

$$\textbf{let } x : T = E \,;\, F$$

Typing rule:

$$(\textsc{Let})\ \frac{\Gamma, x : T \ \vdash\ E : T \qquad \Gamma, x : T \ \vdash\ F : U}{\Gamma \ \vdash\ \textbf{let } x = E \,;\, F : U}$$

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 cannot 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!

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:

$\perp$, cycleF ($\perp$), cycleF(cycleF($\perp$)), ...

What do we get?

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 $\perp$, use a record with fields which are all $\perp$.

Then, the chain is:

$$\{\mathsf{next} = \perp\},\ \mathsf{cycleF}\ \{\mathsf{next} = \perp\},\ \mathsf{cycleF}\ (\mathsf{cycleF}\ \{\mathsf{next} = \perp\})$$

which evaluates to:

$$\{\mathsf{next} = \perp\},\ \{\mathsf{next} = \{\mathsf{next} = \perp\}\},\ \{\mathsf{next} = \{\mathsf{next} = \{\mathsf{next} = \perp\}\}\}$$

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

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.

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.

Let $T = \{f_1 : T_1, \ldots, f_n : T_n\}$.

Then **let** $x : T = E$ translates to:

$$
\begin{array}{l}
\textbf{val}\ x = \{ \\
\quad \begin{array}{llcl}
\textbf{def} & x.f_1\ \&\ \mathit{undef} & = & x.f_1\ \&\ \mathit{defined}(E), \\
 & x.f_1\ \&\ \mathit{defined}(y) & = & y.f_1\ \&\ \mathit{defined}(y), \\
\ldots & & & \\
 & x.f_n\ \&\ \mathit{undef} & = & x.f_n\ \&\ \mathit{defined}(E), \\
 & x.f_n\ \&\ \mathit{defined}(y) & = & y.f_n\ \&\ \mathit{defined}(y)\ ;
\end{array} \\
\quad x\ \&\ \mathit{undef} \\
\}
\end{array}
$$

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 hapens if E selects a field in x?

A Variant

For the purposes of recursive bindings, lazy evaluation is overkill.

It would e 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:

$$\textbf{valrec}\ x : T = E\ ;\ F$$

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

Object Creation

Object Creation can now be expressed 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) ; this
  }
}
```


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?

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).

Example

Consider an abstact 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: ()
}
```


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.

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:

```
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?

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 knows 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.