Foundations of Object-Oriented Programming

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn
Email: `ralf@informatik.uni-bonn.de`
Homepage: `http://www.informatik.uni-bonn.de/~ralf`

February 2007

(Pick up the slides at .../`~ralf/talks.html#54`.)

Vision

Possible future course “Foundations of Programming”:

- FVOP: Foundations of Value-Oriented Programming;
- FEOP: Foundations of Effect-Oriented Programming;
- FOOP: Foundations of Object-Oriented Programming.

☞ A large part of the material is classroom-tested:

- advanced course on “Principles of Programming Languages”;
- introductory course on “Programming”.

Concept

You can never understand one language
until you understand at least two.
Ronald Searle (1920–)

Make everything as simple as possible,
but not simpler.
Albert Einstein (1859–1955)

- *Idea:* explain programming language concepts by growing a "teaching language":
  - empty language,
  - functional language,
  - imperative language,
  - object-oriented language;
- define everything precisely: syntax *and* semantics;
- concentrate on essential concepts and ideas;
- *guiding principle:* a concept should be as simple and pure as possible;
- the concepts should be orthogonal;
- use the teaching language to explain features of "real programming languages".

Foundations of Value-Oriented Programming

$$
\begin{array}{l}
\textbf{local} \\
\quad \textbf{val}\ \mathit{base} = 8 \\
\textbf{in} \\
\quad \textbf{function}\ \mathit{digits}\ (n : \mathit{Nat}) : \mathit{List}\ \langle \mathit{Nat} \rangle = \\
\quad\quad \textbf{if}\ n == 0\ \textbf{then}\ \mathit{Nil} \\
\quad\quad\quad\quad\quad\quad\ \ \textbf{else}\ \ \mathit{Cons}\ (n \mathbin{\%} \mathit{base}, \mathit{digits}\ (n \div \mathit{base})) \\
\quad \textbf{function}\ \mathit{nat}\ (\mathit{ds} : \mathit{List}\ \langle \mathit{Nat} \rangle) : \mathit{Nat} = \\
\quad\quad \textbf{case}\ \mathit{ds}\ \textbf{of} \\
\quad\quad\quad \mathit{Nil} \qquad\qquad\ \ \Rightarrow 0 \\
\quad\quad \mid\ \mathit{Cons}\ (d, \mathit{ds}') \Rightarrow d + \mathit{base} * \mathit{nat}\ (\mathit{ds}') \\
\quad\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

☞ Introduces expressions, declarations, the concept of scope etc.

☞ *Advanced topics:* type abstraction, polymorphism, contracts etc.

Foundations of Effect-Oriented Programming

local
  val $\mathit{bal} = \mathbf{ref}\ 0$
in
  function $\mathit{deposit}\ (\mathit{amount} : \mathit{Nat}) : () =$
    $\mathit{bal} := {!\mathit{bal}} + \mathit{amount}$
  function $\mathit{withdraw}\ (\mathit{amount} : \mathit{Nat}) : () =$
    $\mathit{bal} := {!\mathit{bal}} - \mathit{amount}$
  function $\mathit{balance}\ () : \mathit{Nat} =$
    $!\mathit{bal}$
end

☞ Introduces IO, state, exceptions, the concept of extent etc.

Table of contents

Introduction — Jargon

Every subject has its jargon, object-oriented programming is no exception:

abstract class, anonymous class, behaviour, class hierarchy, class method, class variable, class, constructor, delegation, dispatch table, down cast, dynamic binding, dynamic dispatch, encapsulation, extension, field, final, friend, generic class, implementation, inclusion polymorphism, inheritance, inner class, instance variable, instance, interface inheritance, interface, late binding, message passing, method invocation, method, multiple inheritance, name subtyping, new, object creation, object, object-oriented, open recursion, overriding, package, private, protected, public, redefinition, self, structural subtyping, subclass, subtype polymorphism, subtyping, super, superclass, this, up cast, (virtual) method table, visibility.

☞ Object-orientation seems to be complex and loaded. To help de-mystify the subject, we shall identify the essential principles or characteristics.

Characteristics — What is object-oriented programming?

- Dynamic dispatch:
  When an operation is invoked on an object, the object itself determines what code gets executed. Two objects with the same interface may be implemented quite differently.
- Encapsulation:
  The implementation of an object is hidden from view. Changes to the implementation can only affect the object itself.
- Open recursion:
  One method can invoke another method via a special identifier called *self*, which is late-bound — dynamic dispatch for recursive invocations.
- Subtyping:
  An object that supports more operations can be used as an object that supports less operations. The ability to ignore parts of an interface allows us to write general code that manipulates different sorts of objects in a uniform way.
- Inheritance:
  The behaviour of an object can be reused in another object so that common behaviour must be implemented just once. This reuse of behaviour can be achieved
  - via objects and delegation or
  - via classes and subclassing.

Recapitulation: encapsulation in an effect-oriented language

☞ Encapsulation is achieved by delimiting the scope of internals — only entities that have a name can be accessed and reused.

local
  val *bal* = **ref** 0
in
  function *deposit* (*amount* : *Nat*) : () =
    bal := !*bal* + *amount*
  function *withdraw* (*amount* : *Nat*) : () =
    bal := !*bal* − *amount*
  function *balance* () : *Nat* =
    !*bal*
end

☞ The representation of a bank account, the reference cell *bal*, is local to *deposit*, *withdraw* and *balance*.

☞ *Observation:* The functions *deposit*, *withdraw* and *balance* belong together, but they are only loosely coupled.

Objects

Wish: linguistic support for integrating operations into a single entity.

```
val my-account =
  object
    local
      val bal = ref 0
    in
      method deposit (amount : Nat) : () =
        bal := !bal + amount
      method withdraw (amount : Nat) : () =
        bal := !bal − amount
      method balance : Nat =
        !bal
    end
  end
```

☞ The entity is called an object. As a first approximation an object can be seen as a record of functions.

☞ *Syntactic changes:* **object** . . . **end** bracket; **method** instead of **function**.

Method invocation

☞ An integrated function aka method is invoked using the dot notation. (*Recall:* $\ggg$ is the prompt of the evaluator).

$\ggg$ *my-account.deposit* (4711)
()
$\ggg$ *my-account.withdraw* (815)
()
$\ggg$ *my-account.withdraw* (2765)
()
$\ggg$ *my-account.balance*
1131

Jargon: we say "invoking a method on an object" or "sending an object a message".

Interfaces

☞ Every well-formed expression has a type; **object** . . . **end** is an expression; so what is the type of an object?

$$
\begin{array}{l}
\textbf{type}\ \mathit{Account} = \\
\quad \textbf{object} \\
\qquad \textbf{method}\ \mathit{deposit} \quad : \mathit{Nat} \to () \\
\qquad \textbf{method}\ \mathit{withdraw} : \mathit{Nat} \to () \\
\qquad \textbf{method}\ \mathit{balance} \quad : \mathit{Nat} \\
\quad \textbf{end}
\end{array}
$$

☞ *Recall:* a type represents our static knowledge of an expression.

☞ An object is solely defined by its behaviour; the object's internal representation does not appear in its type!

☞ An object type such as *Account* is also called an interface. An interface is the set of operations an object supports.

Objects and Interfaces

A different implementation of a bank account with the same interface:

$$
\begin{array}{l}
\textbf{val}\ \mathit{your\text{-}account} = \\
\quad \textbf{object} \\
\quad\quad \textbf{local} \\
\quad\quad\quad \textbf{val}\ n \qquad\quad = \textbf{ref}\ 0 \\
\quad\quad\quad \textbf{val}\ \mathit{history} = \textbf{array}\ [10]\ i \Rightarrow \textbf{ref}\ 0 \\
\quad\quad\quad \textbf{function}\ \mathit{inc}\ (n : \mathit{Nat}) : \mathit{Nat} = (n + 1)\ \%\ 10 \\
\quad\quad \textbf{in} \\
\quad\quad\quad \textbf{method}\ \mathit{deposit}\ (\mathit{amount} : \mathit{Nat}) : () = \\
\quad\quad\quad\quad \mathit{history}.[\mathit{inc}\ (!n)] := !\mathit{history}.[!n] + \mathit{amount};\ n := \mathit{inc}\ (!n) \\
\quad\quad\quad \textbf{method}\ \mathit{withdraw}\ (\mathit{amount} : \mathit{Nat}) : () = \\
\quad\quad\quad\quad \mathit{history}.[\mathit{inc}\ (!n)] := !\mathit{history}.[!n] - \mathit{amount};\ n := \mathit{inc}\ (!n) \\
\quad\quad\quad \textbf{method}\ \mathit{balance} : \mathit{Nat} = \\
\quad\quad\quad\quad !\mathit{history}.[!n] \\
\quad\quad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

Demo

```
⋙  my-account.balance
1131
⋙  your-account.deposit 4711
()
⋙  my-account.withdraw 1000; your-account.deposit 1000
()
⋙  my-account.balance
131
⋙  your-account.balance
5711
```

☞ The objects *my-account* and *your-account* are interchangeable — at least from the point of view of the language. They respond to the same messages.

☞ Each object itself determines what code gets executed.

Objects and Interfaces

The fact that objects with the same interface are interchangeable allows us to write code that manipulates objects in a uniform way.

function $\mathit{transfer}\ (\mathit{account}_1 : \mathit{Account}, \mathit{amount} : \mathit{Nat}, \mathit{account}_2 : \mathit{Account}) : () =$
  $\mathit{account}_1.\mathit{withdraw}\ \mathit{amount};$
  $\mathit{account}_2.\mathit{deposit}\quad \mathit{amount}$

☞ We can transfer money between bank accounts, even if the bank accounts are implemented differently.

$\ggg\quad (\mathit{my\text{-}account}.\mathit{balance}, \mathit{your\text{-}account}.\mathit{balance})$
$(131, 5711)$
$\ggg\quad \mathit{transfer}\ (\mathit{your\text{-}account}, 815, \mathit{my\text{-}account})$
$()$
$\ggg\quad (\mathit{my\text{-}account}.\mathit{balance}, \mathit{your\text{-}account}.\mathit{balance})$
$(946, 4896)$

Objects — abstract syntax

We extend our language with anonymous objects and method invocations.

$m \in \text{Method}$		method declarations
$e ::= \cdots$		
	$\mid$ **object** m **end**	anonymous object
	$\mid$ $e.x$	method invocation

☞ An object is a collection of methods.

Methods — abstract syntax

A method declaration is essentially a sequence of method definitions.

$m ::= \textbf{method}\ x : \tau = e$	method definition
$\quad\mid\ m_1\ m_2$	sequential declaration
$\quad\mid\ \textbf{local}\ d\ \textbf{in}\ m\ \textbf{end}$	local declaration

☞ **local** is the interface between the effect-oriented language and the object-oriented language.

☞ **local** makes explicit that the internal representation of an object, possibly comprising instance variables or fields, is local to some of the methods (encapsulation).

Objects — dynamic semantics

☞ The value of an object is essentially a dispatch or method table.

$$\mu \in \mathsf{Id} \to_{\text{fin}} \mathsf{Expr} \qquad \text{method tables}$$

$$\begin{aligned} \nu ::= {} & \cdots \\ \mid {} & \textbf{object}\ \mu\ \textbf{end} \qquad \text{object} \end{aligned}$$

☞ A method table maps a name to an expression, not to a value!

Evaluation rules:

$$\frac{m \Downarrow \mu}{\textbf{object}\ m\ \textbf{end} \Downarrow \textbf{object}\ \mu\ \textbf{end}}$$

$$\frac{e \Downarrow \textbf{object}\ \mu\ \textbf{end} \qquad \mu(x) \Downarrow \nu}{e.x \Downarrow \nu}$$

☞ The name x is looked up at run-time in the method table associated with the object e (dynamic dispatch).

Methods — dynamic semantics

Evaluation rules:

$$\overline{(\textbf{method}\ x : \tau = e) \Downarrow \{x \mapsto e\}}$$

$$\frac{m_1 \Downarrow \mu_1 \qquad m_2 \Downarrow \mu_2}{m_1\ m_2 \Downarrow \mu_1, \mu_2}$$

$$\frac{d \Downarrow \delta \qquad m\delta \Downarrow \mu}{\textbf{local}\ d\ \textbf{in}\ m\ \textbf{end} \Downarrow \mu}$$

☞ Method bodies are not evaluated. Why?

- **method** *balance* : *Nat* is *not* a natural number but a computation that yields a natural number.
- *Later:* the method body may contain the identifier *self*, which refers to the object itself and which is late-bound.

☞ The sequence $m_1\ m_2$ evaluates to the method table μ_1 extended by μ_2; the methods do *not* see each other. Later definitions shadow earlier ones.

☞ The declaration d is local to the method declaration m.

Summary

- An object is solely defined by the set of operations it supports.
- The implementation of an operation is called a method.
- The object's internal representation — state, other objects etc — is hidden from view outside the object's definition (encapsulation).
- When an operation is invoked on an object, the object itself determines what code gets executed (dynamic dispatch).
- An interface is the set of operations an object supports.
- The object's internal representation does not appear in its type.

What's next?

- object templates: classes;
- recursive method invocation: *self*;
- more flexibility: subtyping;
- re-use of behaviours: inheritance via delegation;
- re-use of behaviours: inheritance via classes.

Finite maps

When X and Y are sets $X \rightarrow_{\text{fin}} Y$ denotes the set of finite maps from X to Y. The domain of a finite map φ is denoted $\mathit{dom}\ \varphi$.

- the singleton map is written $\{x \mapsto y\}$
  - $\mathit{dom}\{x \mapsto y\} = \{x\}$
  - $\{x \mapsto y\}(x) = y$
- when φ_1 and φ_2 are finite maps the map φ_1, φ_2 called φ_1 extended by φ_2 is the finite map with
  - $\mathit{dom}\ (\varphi_1, \varphi_2) = \mathit{dom}\ \varphi_1 \cup \mathit{dom}\ \varphi_2$
  - $(\varphi_1, \varphi_2)(x) = \begin{cases} \varphi_2(x) & \text{if } x \in \mathit{dom}\ \varphi_2 \\ \varphi_1(x) & \text{otherwise} \end{cases}$