Introduction

Preliminaries

F-core

I-core

Encapsulation

Subtyping

Inheritance

Conclusion

Appendix: **super**

# Growing an Object-oriented Language

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`

December 2006

(Pick up the slides at .../`~ralf/talks.html#52`.)

# Introduction

A few buzzwords:

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.

# Introduction

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)

- *Goal:* grow a language into an object-oriented language;
- in five (here: three) simple steps.
- Define everything precisely (syntax and semantics).

# Table of contents

Introduction

Preliminaries

The functional core

The imperative core

Encapsulation

Subtyping

Inheritance

Conclusion

Appendix: **super**

# Finite maps



Introduction
Preliminaries
F-core
I-core
Encapsulation
Subtyping
Inheritance
Conclusion
Appendix: **super**

When $X$ and $Y$ are sets $X \to_{\mathrm{fin}} Y$ denotes the set of *finite maps* from $X$ to $Y$. The domain of a finite map $\varphi$ 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 $\varphi_1$ and $\varphi_2$ are finite maps the map $\varphi_1, \varphi_2$ called $\varphi_1$ *extended* by $\varphi_2$ is the finite map with
  - $\mathit{dom}\ (\varphi_1, \varphi_2) = \mathit{dom}\ \varphi_1 \cup \mathit{dom}\ \varphi_2$
  - $(\varphi_1, \varphi_2)(x) = \begin{cases} \varphi_2(x) & \text{if } x \in \mathit{dom}\ \varphi_2 \\ \varphi_1(x) & \text{otherwise} \end{cases}$

# The functional core — example: binary search

```
function binary-search (oracle : Nat → Bool,
                        lower-bound : Nat,
                        upper-bound : Nat) : Nat =
  let
    function search (l : Nat, u : Nat) : Nat =
      if l ⩾ u then u
               else let
                      val m = (l + u) ÷ 2
                    in
                      if oracle m then search (l,     m)
                                  else search (m + 1, u )
                    end
  in
    search (lower-bound, upper-bound)
  end
```

# Local declarations — abstract syntax

Excerpt of the functional core:

$x \in \mathrm{Id}$ *identifiers*

$e \in \mathrm{Expr}$ *expressions*

$$
\begin{aligned}
e ::= {} & \cdots \\
\mid {} & x && \text{identifier} \\
\mid {} & \mathbf{let}\ d\ \mathbf{in}\ e\ \mathbf{end} && \text{local declaration}
\end{aligned}
$$

$d \in \mathrm{Decl}$ *declarations*

$$
\begin{aligned}
d ::= {} & \mathbf{val}\ x = e && \text{value definition/declaration} \\
\mid {} & d_1\ d_2 && \text{sequential declaration} \\
\mid {} & \mathbf{local}\ d_1\ \mathbf{in}\ d_2\ \mathbf{end} && \text{local declaration}
\end{aligned}
$$

# Local declarations — static semantics

$\Sigma \in \mathsf{Sig} = \mathsf{Id} \to_{\mathrm{fin}} \mathsf{Type}$ *signatures*

$$\frac{}{\Sigma \vdash x : \Sigma(x)} \qquad x \in \mathit{dom}\ \Sigma$$

$$\frac{\Sigma \vdash d : \Sigma' \qquad \Sigma, \Sigma' \vdash e : \tau}{\Sigma \vdash \mathbf{let}\ d\ \mathbf{in}\ e\ \mathbf{end} : \tau}$$

$$\frac{\Sigma \vdash e : \tau}{\Sigma \vdash (\mathbf{val}\ x = e) : \{x \mapsto \tau\}}$$

$$\frac{\Sigma \vdash d_1 : \Sigma_1 \qquad \Sigma, \Sigma_1 \vdash d_2 : \Sigma_2}{\Sigma \vdash d_1\ d_2 : \Sigma_1, \Sigma_2}$$

$$\frac{\Sigma \vdash d_1 : \Sigma_1 \qquad \Sigma, \Sigma_1 \vdash d_2 : \Sigma_2}{\Sigma \vdash \mathbf{local}\ d_1\ \mathbf{in}\ d_2\ \mathbf{end} : \Sigma_2}$$

☞ Later definitions shadow earlier ones.



Introduction
Preliminaries
F-core
I-core
Encapsulation
Subtyping
Inheritance
Conclusion
Appendix: **super**

# Local declarations — dynamic semantics

$\delta \in \mathsf{Env} = \mathsf{Id} \rightarrow_{\mathrm{fin}} \mathsf{Val}$ *environments*

$$\frac{d \Downarrow \delta \qquad e\delta \Downarrow \nu}{\textbf{let}\ d\ \textbf{in}\ e\ \textbf{end} \Downarrow \nu}$$

$$\frac{e \Downarrow \nu}{(\textbf{val}\ x = e) \Downarrow \{x \mapsto \nu\}}$$

$$\frac{d_1 \Downarrow \delta_1 \qquad d_2\delta_1 \Downarrow \delta_2}{d_1\ d_2 \Downarrow \delta_1, \delta_2}$$

$$\frac{d_1 \Downarrow \delta_1 \qquad d_2\delta_1 \Downarrow \delta_2}{\textbf{local}\ d_1\ \textbf{in}\ d_2\ \textbf{end} \Downarrow \delta_2}$$

☞ $e\delta$ denotes *substitution*: each free variable $x$ in $e$ with $x \in \mathit{dom}\ \delta$ is replaced by $\delta(x)$.

# The imperative core — example: bank account

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



Introduction
Preliminaries
F-core
I-core
Encapsulation
Subtyping
Inheritance
Conclusion
Appendix: **super**

# References — abstract syntax

Excerpt of the imperative core:

$$
\begin{array}{lll}
e ::= & \cdots & \\
\mid & \mathbf{ref}\ e & \text{creation} \\
\mid & !\ e & \text{dereferencing} \\
\mid & e_1 := e_2 & \text{assignment}
\end{array}
$$

# References — static semantics

$$\tau ::= \cdots$$
$$\quad \mid \; \mathit{Ref}\,\langle\tau\rangle \qquad \text{type of references}$$

$$\frac{\Sigma \vdash e : \tau}{\Sigma \vdash \textbf{ref}\; e : \mathit{Ref}\,\langle\tau\rangle}$$

$$\frac{\Sigma \vdash e : \mathit{Ref}\,\langle\tau\rangle}{\Sigma \vdash\; !e : \tau}$$

$$\frac{\Sigma \vdash e_1 : \mathit{Ref}\,\langle\tau\rangle \qquad \Sigma \vdash e_2 : \tau}{\Sigma \vdash e_1 := e_2 : ()}$$



Introduction
Preliminaries
F-core
I-core
Encapsulation
Subtyping
Inheritance
Conclusion
Appendix: **super**

# References — dynamic semantics



| | | |
|---|---|---|
| $a \in \mathsf{Addr}$ | | *addresses* |
| $\nu ::= \cdots$ | | |
| $\mid a$ | | address |
| $\sigma \in \mathsf{Addr} \to_{\mathrm{fin}} \mathsf{Val}$ | | *store* |

$$\frac{\sigma \mid e \Downarrow \nu \mid \sigma'}{\sigma \mid \textbf{ref}\ e \Downarrow a \mid \sigma', \{a \mapsto \nu\}} \qquad a \notin \mathit{dom}\ \sigma'$$

$$\frac{\sigma \mid e \Downarrow a \mid \sigma'}{\sigma \mid !e \Downarrow \sigma'(a) \mid \sigma'}$$

$$\frac{\sigma \mid e_1 \Downarrow a \mid \sigma_1 \qquad \sigma_1 \mid e_2 \Downarrow \nu \mid \sigma_2}{\sigma \mid e_1 := e_2 \Downarrow () \mid \sigma_2, \{a \mapsto \nu\}}$$

# Encapsulation — example: bank account



```
val bank-account : Bank-Account =
  class
    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
val my-account = new bank-account
```

☞ A *class* is given by an expression.

# Encapsulation — example: bank account



Introduction
Preliminaries
F-core
I-core
Encapsulation
Subtyping
Inheritance
Conclusion
Appendix: **super**

**type** *Bank-Account* =
  **class**
    **method** *deposit* : $Nat \to ()$
    **method** *withdraw* : $Nat \to ()$
    **method** *balance* : $Nat$
  **end**

☞ An *interface* is a type, the type of a class.

# Encapsulation — abstract syntax

| | | |
|---|---|---|
| $e ::= \cdots$ | | |
| $\mid$ | **class** $m$ **end** | anonymous class |
| $\mid$ | **new** $e$ | object creation |
| $\mid$ | $e.x$ | method invocation |

☞ A class is a collection of methods.

| | | |
|---|---|---|
| $m \in$ Method | | *method declarations* |
| $m ::=$ | **method** $x : \tau = e$ | method definition |
| $\mid$ | $m_1\ m_2$ | sequential declaration |
| $\mid$ | **local** $d$ **in** $m$ **end** | local declaration |

☞ Instance variables or fields are locally defined entities (encapsulation).

# Encapsulation — static semantics

$M \in \mathsf{Id} \rightarrow_{\text{fin}} \mathsf{Type}$ *method signatures*

$\tau ::= \cdots$
$\mid$ **class** $M$ **end** class type
$\mid$ **object** $M$ **end** object type

☞ A class type is like an interface.

$$\frac{\Sigma \vdash m : M}{\Sigma \vdash \textbf{class}\ m\ \textbf{end} : \textbf{class}\ M\ \textbf{end}}$$

$$\frac{\Sigma \vdash e : \textbf{class}\ M\ \textbf{end}}{\Sigma \vdash \textbf{new}\ e : \textbf{object}\ M\ \textbf{end}}$$

$$\frac{\Sigma \vdash e : \textbf{object}\ M\ \textbf{end}}{\Sigma \vdash e.x : M(x)} \qquad x \in \mathit{dom}\ M$$

# Encapsulation — static semantics

$$\frac{\Sigma \vdash e : \tau}{\Sigma \vdash (\textbf{method}\ x : \tau = e) : \{x \mapsto \tau\}}$$

$$\frac{\Sigma \vdash m_1 : M_1 \qquad \Sigma \vdash m_2 : M_2}{\Sigma \vdash m_1\ m_2 : M_1, M_2}$$

$$\frac{\Sigma \vdash d : \Sigma' \qquad \Sigma, \Sigma' \vdash m : M}{\Sigma \vdash \textbf{local}\ d\ \textbf{in}\ m\ \textbf{end} : M}$$

☞ Later definitions shadow earlier ones (redefinition).

# Encapsulation — introduction and elimination



Programming language constructs can be divided into introduction and elimination forms.

| | |
|---|---|
| **class** $m$ **end** | introduction |
| **class** $M$ **end** | |
| **new** $e$ | elimination & introduction |
| **object** $M$ **end** | |
| $e.x$ | elimination |

☞ **new** eliminates a class and introduces an object.

# Encapsulation — dynamic semantics

$\mu \in \mathsf{Id} \rightarrow_{\text{fin}} \mathsf{Expr}$ *method environments*

$\nu ::= \cdots$

| **class** *m* **end** — anonymous class

| **object** $\mu$ **end** — object

☞ $\mu$ is a dispatch or method table.

$$\frac{}{\textbf{class}\ m\ \textbf{end} \Downarrow \textbf{class}\ m\ \textbf{end}}$$

$$\frac{e \Downarrow \textbf{class}\ m\ \textbf{end} \qquad m \Downarrow \mu}{\textbf{new}\ e \Downarrow \textbf{object}\ \mu\ \textbf{end}}$$

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

☞ Evaluation of classes is delayed; **new** triggers it; evaluation of methods is delayed (next slide); method invocation triggers it (dynamic dispatch).

# Encapsulation — dynamic semantics



$$\frac{}{(\textbf{method}\ x : {\color{red}\tau} = e) \Downarrow \{x \mapsto e\}}$$

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

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

☞ Methods 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 bound late.

# Encapsulation — live cycles of classes and objects



```
val bank-account : Bank-Account =
  class local val bal = ref 0
        in    method deposit (amount : Nat) : () =
                bal := !bal + amount
   ... end end
```

☞ *bank-account* is bound to the class expression; the evaluation of **class** ... **end** is delayed.

```
val my-account = new bank-account
```

☞ *my-account* is bound to an instance of *bank-account*; **new** triggers the evaluation of the class: a new reference cell is allocated. The evaluation of the method *deposit* is delayed.

```
my-account.deposit (4711)
```

☞ The method *deposit* is evaluated: $4711$ is added to the contents of *bal*.

# Encapsulation — constructors



Introduction
Preliminaries
F-core
I-core
Encapsulation
Subtyping
Inheritance
Conclusion
Appendix: **super**

```
function bank-account (money : Nat) : Bank-Account =
  class
    local
      val bal = ref money
    in
      method deposit (amount : Nat) : () =
        bal := !bal + amount
      method withdraw (amount : Nat) : () =
        bal := !bal − amount
      method balance : Nat =
        ! bal
    end
  end
val my-account = new (bank-account 4711)
```

☞ A constructor is a class-valued expression (often a function).

# Encapsulation — class variables and methods

**local**
  **val** *no* $=$ **ref** $0$
**in**
  **function** *number-of-accounts* $()$ : *Nat* $=$
    $!\,no$
  **val** *bank-account* $=$
    **class**
      **local**
        **val** *bal* $= no := !no + 1;$ **ref** $0$
      **in**
        **method** *deposit* $(amount : Nat) : () =$
          $bal := !bal + amount$
          $\ldots$
      **end**
    **end**
**end**

☞ A class variable is introduced outside the class; a class method is a function.

# Subtyping — static semantics



*Observation:* $e.x$ is valid if $e$ has at least an $x$ method.

$$\frac{\Sigma \vdash e : \tau \qquad \tau \preccurlyeq \tau'}{\Sigma \vdash e : \tau'}$$

$$\frac{M(x) \preccurlyeq M'(x) \quad | \quad x \in \mathit{dom}\ M}{\mathbf{class}\ M\ \mathbf{end} \preccurlyeq \mathbf{class}\ M'\ \mathbf{end}} \qquad \mathit{dom}\ M' \subseteq \mathit{dom}\ M$$

$$\frac{M(x) \preccurlyeq M'(x) \quad | \quad x \in \mathit{dom}\ M}{\mathbf{object}\ M\ \mathbf{end} \preccurlyeq \mathbf{object}\ M'\ \mathbf{end}} \qquad \mathit{dom}\ M' \subseteq \mathit{dom}\ M$$

☞ "Width and depth subtyping" also known as interface inheritance.

☞ Inclusion or subtype polymorphism.

# Inheritance — example: extended bank account

```
val extended-bank-account =
  class
    inherit bank-account
    method clear : Nat =
      let
        val money = self.balance
      in
        self.withdraw money;
        money
      end
  end
```

☞ Only methods are inherited; no information leakage via inheritance.

☞ Multiple inheritance via multiple **inherit** declarations.

# Inheritance — abstract syntax

$m ::= \cdots$
$\quad \mid\ \mathbf{inherit}\ e$ inherit from a class

☞ *In addition:* method bodies may contain the identifier *self* (this in other languages).

☞ *self* introduces recursion (open recursion).

# Inheritance — static semantics

$$\frac{\Sigma \mid \textbf{object}\ M\ \textbf{end} \vdash m : M}{\Sigma \vdash \textbf{class}\ m\ \textbf{end} : \textbf{class}\ M\ \textbf{end}}$$

$$\frac{\Sigma, \{self \mapsto \gamma\} \vdash e : \tau}{\Sigma \mid \gamma \vdash (\textbf{method}\ x : \tau = e) : \{x \mapsto \tau\}}$$

$$\frac{\Sigma \vdash d : \Sigma' \qquad \Sigma, \Sigma' \mid \gamma \vdash m : M}{\Sigma \mid \gamma \vdash \textbf{local}\ d\ \textbf{in}\ m\ \textbf{end} : M}$$

$$\frac{\Sigma \vdash e : \textbf{class}\ M\ \textbf{end}}{\Sigma \mid \gamma \vdash \textbf{inherit}\ e : M}$$

☞ The class type is needed to check the class expression.

☞ *self* is only visible in the method bodies. Why?

- *self* is only bound when a method is invoked (late binding).




Introduction

Preliminaries

F-core

I-core

Encapsulation

Subtyping

Inheritance

Conclusion

Appendix: **super**

# Inheritance — dynamic semantics

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

$$\frac{e \Downarrow \textbf{class}\ m\ \textbf{end} \qquad m \Downarrow \mu}{\textbf{inherit}\ e \Downarrow \mu}$$

☞ *self* is bound to the object on which the method was invoked.

☞ **inherit** is similar to **new**.

☞ *Visibility*: there is no difference between **new** and **inherit**. This is a conscious design decision.



Introduction
Preliminaries
F-core
I-core
Encapsulation
Subtyping
Inheritance
Conclusion
Appendix: **super**

# Conclusion

- Object-orientation in five (here: three) simple steps.
- The resulting language is fairly expressive.
- Classes are first-class values.
- Classes with a single instance: **new class** . . . **end**.
- Generic classes via polymorphic constructor functions (no extension required).



Introduction
Preliminaries
F-core
I-core
Encapsulation
Subtyping
Inheritance
Conclusion
Appendix: **super**

# **super** — example: bank account with a memory

```
val max-bank-account =
  class
    inherit
      super = extended-bank-account
    in
      local
        val max = ref (super.balance)
      in
        method deposit (amount : Nat) : () =
          super.deposit amount;
          max := maximum (!max, super.balance)
        method max-balance : Nat =
          ! max
      end
    end
  end
```

# **super** — abstract syntax

$m ::= \cdots$
$\quad \mid \ \textbf{inherit}\ x = e\ \textbf{in}\ m\ \textbf{end}$ Vererbung

☞ $x$ is the name of the superobject; $e$ is the superclass.

☞ By the way: overriding is simple; just define a new method with the same name.

# **super** — static semantics

$$\frac{\Sigma \vdash e : \textbf{class}\ M_1\ \textbf{end} \qquad \Sigma, \{x \mapsto \textbf{object}\ M_1\ \textbf{end}\} \mid \gamma \vdash m : M_2}{\Sigma \mid \gamma \vdash \textbf{inherit}\ x = e\ \textbf{in}\ m\ \textbf{end} : M_1, M_2}$$

# super — dynamic semantics

$$
\begin{array}{lll}
\nu ::= \cdots & & \\
\quad \mid \ \textbf{super}\ \mu\ \textbf{of}\ \nu\ \textbf{end} & & \text{superobject including `}\mathit{self}\text{'}
\end{array}
$$

$$
\frac{e \Downarrow \textbf{class}\ m_1\ \textbf{end} \qquad m_1 \Downarrow \mu_1 \qquad m\{x \mapsto \textbf{super}\ \mu_1\ \textbf{of}\ \mathit{self}\ \textbf{end}\} \Downarrow \mu_2}{\textbf{inherit}\ x = e\ \textbf{in}\ m\ \textbf{end} \Downarrow \mu_1, \mu_2}
$$

$$
\frac{e \Downarrow \textbf{super}\ \mu\ \textbf{of}\ o\ \textbf{end} \qquad \mu(x)\{\mathit{self} \mapsto o\} \Downarrow \nu}{e.x \Downarrow \nu}
$$

☞ Tricky!

- $x$ is bound to the superobject;
- the second field of **super** $\mu$ **of** $\nu$ **end** is set to *self*;
- when a method of the subobject is invoked, *self* is bound;
- this object is then passed to each super call.