A Core Calculus of Mixin-Based Incomplete Objects

GC Meeting, Venezia, June 2004

Lorenzo Bettini[1], Viviana Bono[2], Silvia Likavec[2]

[1]Dip. di Sistemi e Informatica, Univ. di Firenze
[2]Dip. di Informatica, Univ. di Torino

Rationale

- «Ideally, you shouldn't have to create new components to achieve reuse. You should be able to get all the functionalities you need just by assembling existing components through object composition [...]» [*Design Patterns*]

  Favor object composition over class inheritance

- Sometimes inheritance is (wrongly) used instead of object composition
- Design patterns may help, but they are often a "programming patch" to a feature that is missing in the programming language

Mixin...

- ... a class definition parameterized over the superclass
- ... a function that takes a class as an argument and produces another (sub)class
- ... minimizes code dependencies
  - A subclass can be implemented before a superclass
  - The same mixin can be applied to many superclasses

Our approach

- Extend the calculus of classes and mixins [BonoPatelShmatikov99] with incomplete objects
- We can:
  - apply a mixin to a class to obtain a subclass
  - instantiate a class to create an object
  - instantiate a mixin to produce an incomplete object
  - complete incomplete objects to obtain a complete object

The syntax of the core calculus

$$
\begin{array}{rcl}
e & ::= & \mathit{const} \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathit{fix} \mid \mathsf{ref} \mid\ ! \mid :=\\
& \mid & \{x_i = e_i\}^{i\in I} \mid e.x \mid \mathsf{H}\ h.e \mid \mathsf{new}\ e\\
& \mid & \mathsf{classval}\langle v_g, \mathcal{M}\rangle \mid e_1 \diamond e_2\\
& \mid & e_1 \longleftarrow\!\!\!+\ m_i = e_2 \mid e_1 \longleftarrow\!\!\!+\ e_2\\
& \mid & \mathsf{mixin}\\
& & \quad \mathsf{method}\ m_j = v_{m_j};\ ^{(j\in New)}\\
& & \quad \mathsf{redefine}\ m_k = v_{m_k};\ ^{(k\in Redef)}\\
& & \quad \mathsf{expect}\ m_i;\ ^{(i\in Expect)}\\
& & \quad \mathsf{constructor}\ v_c;\\
& & \mathsf{end}
\end{array}
$$

Some details

- An extension of a functional calculus with side effects
- All of the methods are function of one private field and of *self*:
  λ*myfield*. λ*self*. `...body...`
- Overriding methods are also function of *next*, that can be used to access the superclass implementation of the method:
  λ*myfield*. λ*self*. λ*next*. `...body...`
- The constructor takes an argument and returns:
  - the value to initialize the private field
  - the argument to pass to the superclass constructor

Mixins

- A mixin is made of
  - *defined* methods
    method $m_j = v_{m_j};$ $^{(j \in New)}$
  - *redefined* methods
    redefine $m_k = v_{m_k};$ $^{(k \in Redef)}$
  - *expected* methods
    expect $m_i;$ $^{(i \in Expect)}$
  - a *constructor*
    constructor $v_c;$

Mixin application

- A mixin m can be applied to a class c that provides:
  - all of the methods *expected* by the mixin
  - all of the methods that a mixin expects to *redefine*
- The mixin application $m \diamond c$ will generate a new (sub)class with:
  - all of the methods defined (and redefined) by the mixin and
  - all of the methods defined by the class (and not redefined by the mixin)

Root class

- We define the root of the class hierarchy, class *Object*, as a predefined class
- Only mixins can be written
- A user-defined class can be obtained by applying a mixin with all defined methods (no expectations) to *Object*

$$\begin{array}{l}\text{class}\\ \ \text{method } m_j = v_{m_j};\\ \ \text{constructor } v_c;\\ \text{end}\end{array} \equiv \left(\begin{array}{l}\text{mixin}\\ \ \text{method } m_j = v_{m_j};\\ \ \text{constructor } v_c;\\ \text{end}\end{array} \diamond \mathit{Object}\right)$$

(Complete) objects

- A (complete) object can be created by instantiating a class and passing an argument to the constructor:
  $\mathsf{new}(m \diamond c)\ myarg$
- All of its methods can be invoked:
  $\mathsf{let}\ o = \mathsf{new}(m \diamond c)\ myarg\ \mathsf{in}\ o.m\ x$

Incomplete objects

- An incomplete object can be created by instantiating a mixin
  new $m\ myarg$
- Only methods that are “complete” can be invoked (those that are not expected and in turn do not use expected methods)
- Can be completed:
  - one method at time, via *method addition*:
    $o \longleftarrow\!\!+\ m_i = v_i$
  - in one step, via *object composition* (with a complete object):
    $o \longleftarrow\!\!+\ o'$

Method addition

- It can be applied only to incomplete objects
- The added method is parameterized over *self*:
  - it can make type assumptions on *self*
  - it can call methods on *self* (once it is added to an incomplete object)
- Statically checked by the type system
- Thus, when a method is added it becomes an effective component of the host object:
  - not only the methods of the object can invoke the new added method
  - but also the new added method can use any of its sibling methods

Object composition

- It is the “transposition” of mixin application to the object level
- An incomplete object can be completed with a complete object that has:
  - all of the methods that are missing in the incomplete object (the mixin *expected* methods)
  - all of the methods that the incomplete object expects to *redefine* (the mixin *redefined* methods)
- Dynamic binding will be applied for *redefined* methods, thus also the *self* of the complete object will be updated
- Not just syntactic sugar for many method additions (the object has a state)

Object completion via *method addition*

A scenario

- Sometimes it is desirable to add some functionalities to existing objects without creating new mixins only for this purpose:
  - consider the development of a graphical application that uses widgets such as buttons, menus and keyboard shortcuts
  - these widgets are associated an event listener (callback function) that is triggered upon specific events (e.g., mouse click)
- You only need to add a function to an existing object

The *command* design pattern

- «Encapsulate a request as an object, thereby letting one parameterize clients with different requests»
- Allows to parameterize a widget over the event handler
- The same event handler can be reused for similar widgets
  - E.g., the event handler “save file” can be associated with the “save” button, with the “save” menu item and with the keyboard shortcut `Ctrl+S`

Drawbacks of the pattern

- One must manually program the pattern
- One must create a class for the command, while a simple function would do
- One must check that the handler is associated at run-time (to avoid “null pointer” problem)

Widget mixins

```
let Button =
  mixin
    method display
    method setEnabled
    expect onClick
    ...
  end in
let MenuItem =
  mixin
    method show
    method setEnabled
    expect onClick
    ...
  end in
let ShortCut =
  mixin
    method setEnal
    expect onClick
    ...
  end in
    let ClickHandler =
      λ self. ... doc.save() ... self.setEnabled(false)
      in
       let button = new Button("Save") in
       let item = new MenuItem("Save") in
       let short = new ShortCut("Ctrl+S") in
         button ←+ (OnClick = ClickHandler);
         mydialog.addButton(button); // now it is safe to use it
         item ←+ (OnClick = ClickHandler);
         mymenu.addItem(item);
         short ←+ (OnClick = ClickHandler);
         system.addShortCut(short);
```

Advantages

- The system is implemented through language constructs (we do not need to bother to manually implement the command pattern class structures)
- The correct use is statically type-checked:

  button $\longleftarrow\!\!\!+$ (OnClick = ClickHandler);
  mydialog.addButton(button);
  item $\longleftarrow\!\!\!+$ (OnClick = ClickHandler);
  mymenu.addItem(item);
  short $\longleftarrow\!\!\!+$ (OnClick = ClickHandler);
  system.addShortCut(short);

  These methods require complete objects and the type system “knows” that at this point the objects are complete

Advantages (cntd.)

- The same listener can be simply installed to more incomplete objects (ensuring consistency in the application)
- The added method can rely on methods of the host object:

  $$\lambda\, \textit{self}\,.\ \ldots\ \text{doc.save()}\ \ldots \textit{self}\,\text{.setEnabled(}\texttt{false}\text{)}$$

  The type system will check that the host object provides this method

Widget mixins (cntd.)

```
let FunnyButton =
  mixin
    method display
    method setEnabled
    method playSound
    redefine onClick =
    λself. λnext. ...next() ...
      self.playSound("tada.wav");
  end in

let funnybutton = new FunnyButton("Save")  in
  funnybutton.display();
  funnybutton ←+ (OnClick = ClickHandler);
  toolbar.addButton(funnybutton);
```

Object completion via *object composition*

Object composition & aggregation

- It is often advocated as a powerful alternative to class inheritance in that it is defined at run-time and it enables dynamic object code reuse by assembling existing components
- Used in conjunction with *delegation* to forward method requests to other objects
- You build a “chain” or “cascade” of objects (an object has a sub-object to whom it delegates a method invocation after performing some actions)
- It is often the right alternative to some forms of multiple inheritance

The *decorator* design pattern

- «Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionalities.»
- Used in implementing stream in class libraries:
  - a base class defines the stream interface
  - some final derived classes provides specific stream functionalities (e.g., file streams, network streams, etc.)
  - other derived classes add functionalities (e.g., buffering, compression, etc.) and their objects can be composed with objects of stream classes

Stream mixins

```
let File =
  mixin
   method write = . . .
   method read = . . .
   . . .
  end in

let Socket =
  mixin
    method write = . . .
    method read = . . .
    method IP = . . .
    . . .
  end in

let Console =
  mixin
    method write = . . .
    method read = . . .
    method setFont = . . .
    . . .
  end in

let Compress =
  mixin
    redefine write = λlevel. λself. λnext. λdata.  next (compress(data
    redefine read = λlevel. λself. λnext. λ_ .  uncompress(next (), lev
    constructor λ(level, arg). {fieldinit=level, superinit=arg};
  end in . . .
let Buffer =
  mixin
    redefine write = λsize. λself. λnext. λdata.  // bufferize write reque
    redefine read = λsize. λself. λnext. λ_ .  // read from the buffer;
    constructor λ(size, arg). {fieldinit=size, superinit=arg};
  end in . . .
```

Using streams as incomplete objects

- We can create a stream that writes into a compressed file by completing a `Compress` object with a `File` object:

$$\begin{array}{l} \textsf{let fileoutput =} \\ \quad (\textsf{new Compress}(\texttt{"HIGH"})) \longleftarrow\!\!\!+ \\ \quad\; (\textsf{new (File} \diamond \textit{Object}\textsf{)} \; (\texttt{"foo.txt"})) \;\; \textsf{in} \\ \quad\;\; \textsf{fileoutput.write}(\texttt{"bar"}) \end{array}$$

- We can also create a chain of streams:

$$\begin{array}{l} \textsf{let fileoutput =} \\ \quad (\textsf{new UUEncode}(\texttt{"base64"})) \longleftarrow\!\!\!+ \\ \quad\; (\textsf{new Compress}(\texttt{"HIGH"})) \longleftarrow\!\!\!+ \\ \quad\; (\textsf{new Buffer}(\texttt{1024})) \longleftarrow\!\!\!+ \\ \quad\;\; (\textsf{new (File} \diamond \textit{Object}\textsf{)} \; (\texttt{"foo.txt"})) \;\; \textsf{in} \\ \quad\;\;\; \textsf{fileoutput.write}(\texttt{"bar"}) \end{array}$$

Using streams as incomplete objects

- The same additional functionalities (compression, buffering) can be applied also to other basic streams such as sockets (since `Socket` is able to fulfill all the expectations of incomplete objects):

```
let outsocket =
  (new Compress("HIGH")) ←+
   (new (Socket ◇ Object) ("192.168.0.71:8080")) in
    outsocket.write("GET foo")

let outsocket =
  (new UUEncode("base64")) ←+
   (new Compress("HIGH")) ←+
   (new Buffer(1024)) ←+
    (new (Socket ◇ Object) ("192.168.0.71:8080")) in
     outsocket.write("GET foo")
```

Logging functionalities

- We can program a `Logger` that is parameterized over a stream:

```
let Logger =
  mixin
    method doLog = λ verb. λ self. λ msg.
      write(self.getTime() + ":    " + msg);
    method getTime = ...
    expect write;
    ...
  end in

let logger = new Logger(verbosity) ←+ output  in
 output.doLog("logging started...");
 output.doLog("log some actions...");
```

where `output` must provide at least `write` (can be either `fileoutput` or `outsocket` seen before)

A multiplexer

- We can further exploit object composition:

```
let Multiplexer =
  mixin
    method addTarget =
      // add the target to the list;
    method removeTarget =
      // remove the target from the list;
    method write =
      // call "write" on every object in the list
    ...
  end in

let multi = new Multiplexer ◇ Object in
  multi.addTarget(fileoutput);
  multi.addTarget(outsocket);
  let logger = new Logger(verbosity) ⟵+ multi  in
    output.doLog("logging started...");
    output.doLog("log some actions...");
```

Type System & Properties

Mixin types

mixin$\langle \gamma_b, \gamma_d, \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old} \rangle$

- γ_b, expected argument of the superclass generator
- γ_d, argument of the mixin generator
- $\Sigma_{new} = \{m_j : \tau^{\downarrow}_{m_j}\}$, methods introduced by the mixin
- $\Sigma_{red} = \{m_k : \tau^{\downarrow}_{m_k}\}$, methods redefined by the mixin
- $\Sigma_{exp} = \{m_i : \tau^{\uparrow}_{m_i}\}$, methods that are expected to be supported by the superclass
- $\Sigma_{old} = \{m_k : \tau^{\uparrow}_{m_k}\}$, types of *next*'s

Mixin application

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old} \rangle \\ \Gamma \vdash e_2 : \mathsf{class}\langle \gamma_c, \Sigma_b \rangle \\ \Gamma \vdash \gamma_b <: \gamma_c \\ \Gamma \vdash \Sigma_b <: (\Sigma_{exp} \cup \Sigma_{old}) \\ \Gamma \vdash \Sigma_{red} <: \Sigma_b / \Sigma_{old} \\ Subj(\Sigma_b) \cap Subj(\Sigma_{new}) = \emptyset \end{array}}{\Gamma \vdash e_1 \diamond e_2 : \mathsf{class}\langle \gamma_d, \Sigma_d \rangle} \quad (\textit{mixin app})$$

where $\Sigma_d = \Sigma_{new} \cup \Sigma_{red} \cup (\Sigma_b - (\Sigma_b / \Sigma_{red}))$

Incomplete object type

obj$\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old} \rangle$

- $\Sigma_{new} = \{m_j : \tau^{\downarrow}_{m_j}\}$, methods introduced by the mixin
- $\Sigma_{red} = \{m_k : \tau^{\downarrow}_{m_k}\}$, methods redefined by the mixin
- $\Sigma_{exp} = \{m_i : \tau^{\uparrow}_{m_i}\}$, methods that are to be provided through method addition or object composition
- $\Sigma_{old} = \{m_k : \tau^{\uparrow}_{m_k}\}$, types of *next*'s
- no information about constructors since an incomplete object has already been initialized (the private field is already bound)

Method addition

$$\frac{\begin{array}{l}\Gamma \vdash e : \mathsf{obj}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old} \rangle \\ m_i : \tau^{\uparrow}_{m_i} \in \Sigma_{exp} \\ \Gamma \vdash \tau_{m_i} <: \tau^{\uparrow}_{m_i} \\ \Gamma \vdash v_{m_i} : \Sigma_1 \longrightarrow \tau_{m_i} \\ \Gamma \vdash (\Sigma_{new} \cup \{m_i : \tau_{m_i}\} \cup \Sigma_{red} \cup \Sigma_{exp} - \{m_i : \tau^{\uparrow}_{m_i}\}) <: \Sigma_1\end{array}}{\begin{array}{l}\Gamma \vdash e \longleftarrow\!\!+ \ (m_i = v_{m_i}) : \\ \qquad \mathsf{obj}\langle \Sigma_{new} \cup \{m_i : \tau_{m_i}\}, \Sigma_{red}, \Sigma_{exp} - \{m_i : \tau^{\uparrow}_{m_i}\}, \Sigma_{old} \rangle\end{array}}$$

Rule for addition of a method that will be “redefined” is similar

Method invocation

$$\frac{\Gamma \vdash e : \mathsf{obj}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp}, \Sigma_{old} \rangle \quad m_i : \tau_{m_i} \in \Sigma_{new} \quad \mathit{TransDep}(m_i) \subseteq \mathit{Subj}(\Sigma_{new})}{\Gamma \vdash e.m_i : \tau_{m_i}}$$

A method m on an incomplete object can be invoked provided the method is “complete” (defined in the object) and all of the methods called by m are complete, recursively.

Properties

Subject Reduction (reduction preserves types)

If $\Gamma \vdash e : \tau$ and $e \twoheadrightarrow e'$, then $\Gamma \vdash e' : \tau$.

Soundness (under the condition that a program terminates, if the program is well-typed, it will not "get stuck", i.e., no "message-not-understood" error will occur during the computation)

Let p be a program: if $\varepsilon \vdash p : \tau$ then either $p \Uparrow$ or $p \mapsto\!\!\!\rightarrow v$ and $\varepsilon \vdash v : \tau$, for some value v.

Conclusions

- A tradeoff between flexibility and static type safety
- It provides two linguistic features that are usually emulated through manual programming (*command* and *decorator* patterns)
- At the moment we do not have subtyping even on complete objects (subtyping/inheritance conflicts): **WORK-IN-PROGRESS** [RieckeStone2002, BeBoVe2004]
- Study possible other further extensions:
  - *higher-order* mixins (i.e., mixins that can be composed with other mixins) and a more general object composition operation, that matches mixin composition
  - an object-based method override