Re-exploring multiple inheritance.

Didier Rémy
INRIA Rocquencourt

FOOL'7

Modular programming

Module systems

- control of the visibility of fields
- type abstraction and generativity
- functor abstraction and application

⇒ **weak** modularity: code reuse must be thought of in advance.

Object orientation

- abstraction by control of the visibility of fields.
- adaptability: late binding and method overriding.
- extensibility: method extension.

⇒ **strong** modularity: code reuse can be done a posteriori.

Two paradigms for objects

⇒ **Objects** as record of functions.

Smalltalk, C++, Java, Eiffel, Ocaml, Moby — Most calculi

Methods as (dynamically resolved) overloaded functions.
CLOS, Cecil, Jazz (ML-sub) — Castagna's ML&

Inheritance, the key to modularity

Single inheritance

- a single class can be refined at once.
- all kinds of operations can be considered: changing visibility attributes, overriding, extension, renaming, ...
- but only method by method, and explicitly.

The inheritance is tree-like, which limits the possible reuse of existing classes.

Multiple inheritance

- includes all features of single inheritance, and
- allows merging two existing classes together.

Multiple inheritance

A popular concept, often mentioned, but too often excluded from calculi and language designs...

Available in C++, Eiffel, Ocaml, Cecil, (+ mixins)

All have deficiencies

- Some of them are insufficiently formalized.
- Most of them have a bad or restrictive overriding policy.
- All but Eiffel are limited by lack of renaming capabilities.

However...

- The union of existing languages contain interesting ideas.
- Multiple inheritance remains useful.

Backups

A class provides superficial backup for all its instance variables.

```
class backup =
  object (self)
    val mutable last = None
    method save = last <- Some (Oo.copy self)
    method restore = match last with (Some x) -> x | _ -> self
  end;;
```

(Obvious variations on the depth of saving, etc.*)*

Such a class can be added to (almost) any existing class.

```
class counter_with_backups =
  object inherit counter inherit backup end
```

Modular components

Enriching a class by adding optional modular components.

Many possible variations on this theme.

```
class counter =
  object val mutable x = 0 method incr = x <- succ x end;;
class button action =
  object method mouse : unit = action self end;;

class counter_with_button =
  object (self)
    inherit counter
    inherit button (fun self -> self#incr)
  end;;
```


The alternative

Use a button object instead of inheriting from the button class:

```
class counter_with_button =
  object (self)
    inherit counter
    val b = new button (fun z -> self #incr)
    method button = b#button
  end;;
```

However, the button is closed (an object, no more a class)

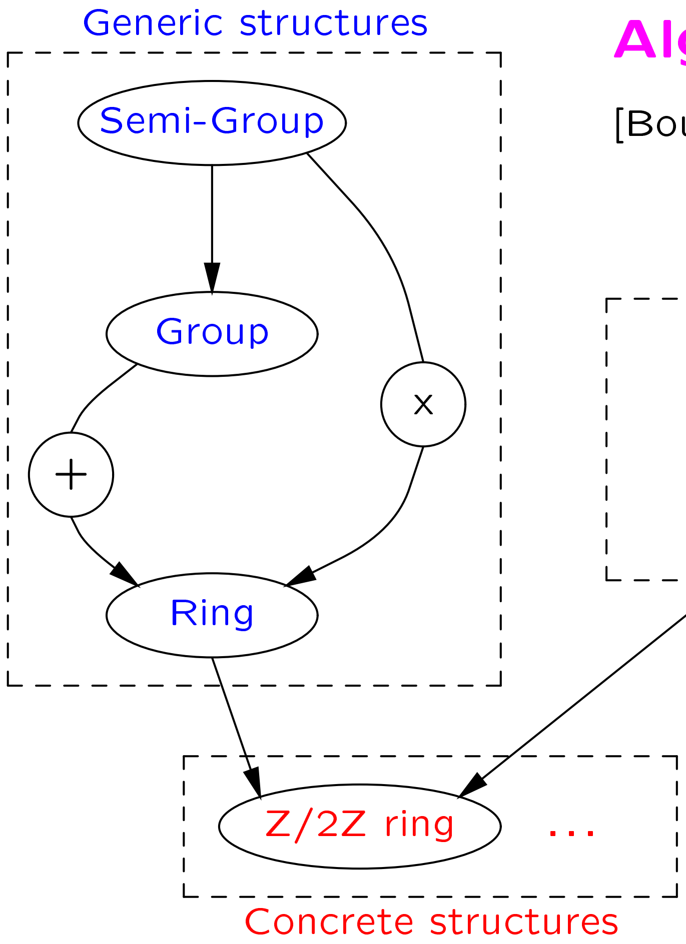
- It cannot intimately interact with the counter, by sharing the same, internal view of self.

  Small initialization problem (`self` is ill-typed) that can be fixed, but with a more complicated connection protocol.

- Its behavior cannot be refined in a subclass.

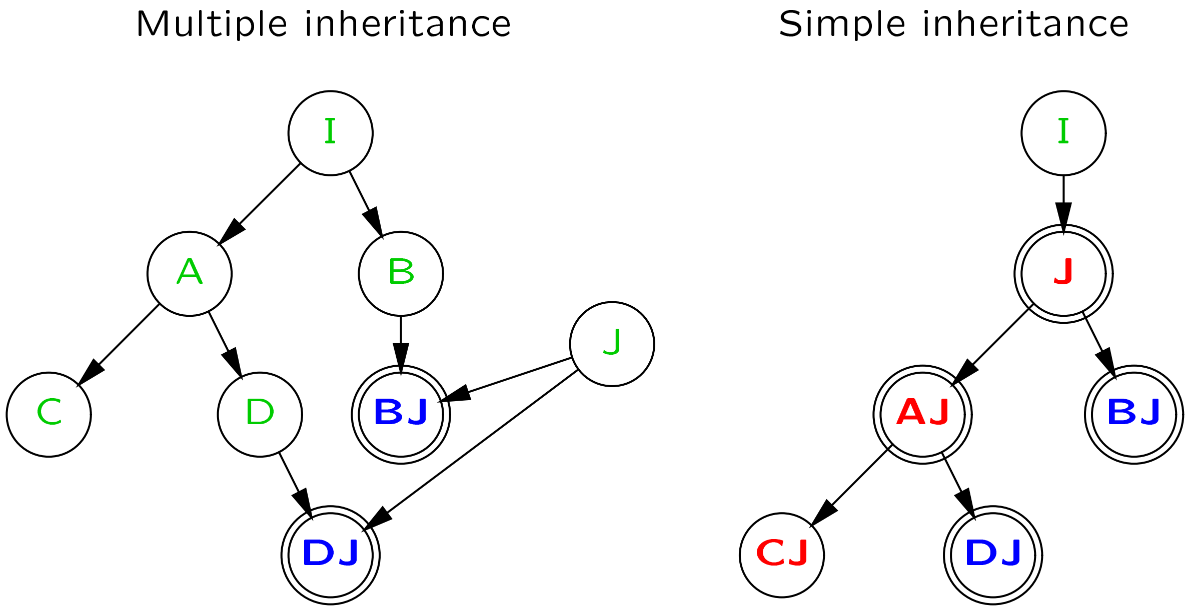
Algrebraic structures

[Boulmé,Hardin, Rioboo at Paris 6]



- Multiple inheritance,
- Late binding,
- No state encapsulation.
- Modules, instead.

Multiple v.s. single inheritance



Classes B and D are extended by J, a posteriori, without rewriting any of them.

All other classes but I and CJ must be modified. All classes but I are affected by class J.

Wrappers

Wrappers are sometimes used in place of inheritance.

A wrapper is a function taking an hypothetical parent class as argument that can be extended using single inheritance.

Backups as a wrapper

```
wrapper backup (class c : classtype ct end) =
  object (self)
    inherit c
    val mutable last = None
    method save = ...
    method restore = ...
  end
```

The wrapper can be applied to the class `counter`:

```
class counter_with_backup = backup (counter)
```


Weaknesses of wrappers

Lost of information (small typing problem)

The wrapper assumes some input interface for class `c`. It can be applied to a class with a richer interface, but the extra components will be hidden in the resulting class, unless an **expressive** type system with a form of matching is used.

Weak form of modularity:

- `backup` must have been defined as a wrapper from start.
- Methods that are overridden by a wrapper must be known at the definition of the wrapper.

Functors provide wrappers (with restrictions above)

A complement to inheritance

Wrappers are not a replacement to M.I.

- they are convenient when several classes must be extended in the same way with some new features J.
- they are useless if J is already defined and compiled.

Multiple inheritance can still benefit from wrappers.

See also

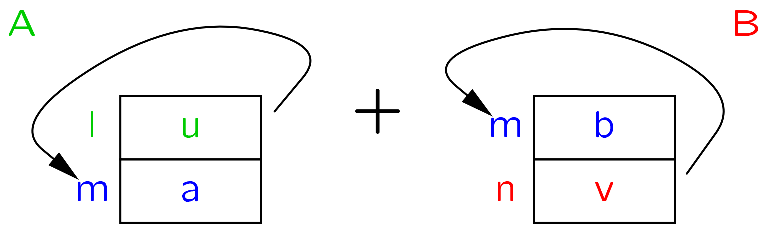
[Bracha-Cook 90] mixin based inheritance.

[MLART] (precursor of Ocaml): wrappers replaced multiple inheritance and were typed with row variables.

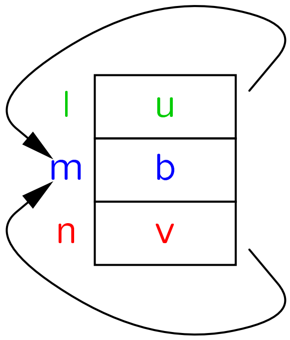
[Flinder-Flatt, Fisher-Reppy] adding mixins to Java and Moby

The difficult problem with M.I.

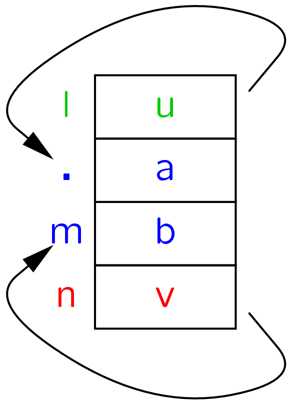
When a method m has implementations a and b in classes A and B, should one definition override or hide the other?



Overriding m



Hiding m



Examples not to follow

Of course, picking either strategy, once for all, is bad.

Ocaml's policy as a counter-example

- public methods are shared and overridden.
- Private names are hidden and duplicated.

Whether a repeated method should be hidden or overridden is independent of its visibility in the final object.

The challenge is to find a good and expressive policy

A false problem

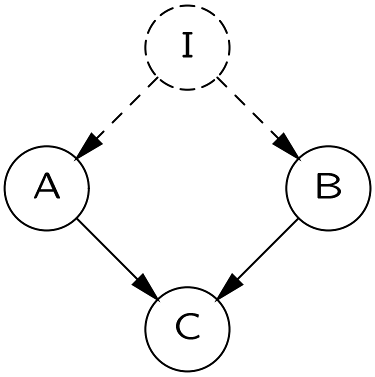
Once hiding or overriding has been chosen, which definition should be given priority?

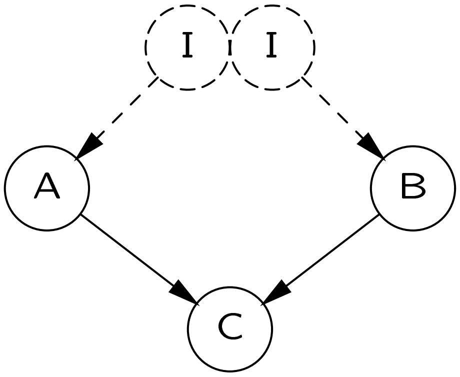
- Textual order is usually fine.
- If not, the conflict can be eliminated by explicitly hiding or deleting a method definition on either side.

In Cecil, there is no default priority and the user must always manually prevent conflicts.

A difficult case

Multiple inheritance of a common antecedent



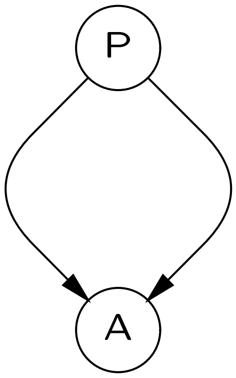


Should I be shared or duplicated in C?

- Shouldn't the two versions be undistinguishable (in the definition of C)?
- A share antecedent must not necessarily be shared in C, and conversely.

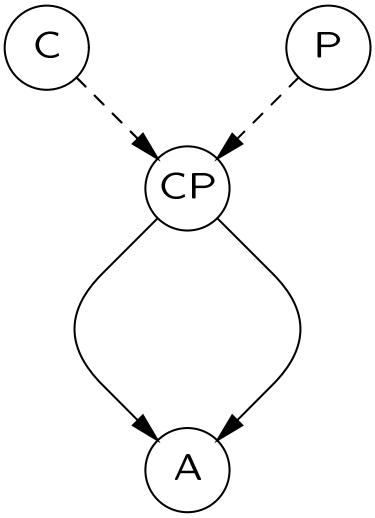
And yet more difficult...

Arrows (A)
as pairs of points (P)



P should be split in A.

Arrows with color (C)

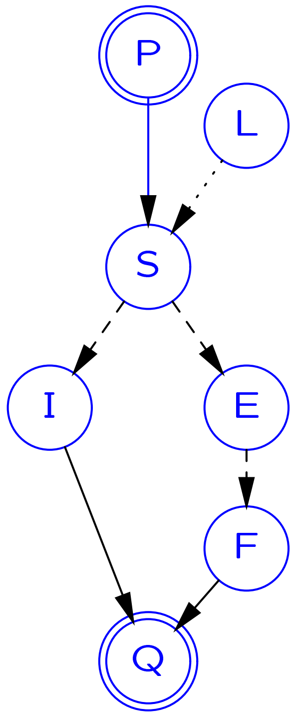


P should have two independent copies in A, but C should/could be shared.

An example in concurrency

Given a parent class P where methods run concurrently, define a version Q where external calls are sequential, i.e. *are locked while internals recursive calls are executed.*

A solution in the Objective join-calculus



- Derive S from P controlling all methods by a lock L
- Derive two copies E and I of P.
- Modify E so that all methods switch from internal to external states on entering and exiting.
- Merge E and I together in Q:
  - State components are shared,
  - Locks and methods are split,
  - Methods of I are renamed and private,
  - Methods of E are public.

Our challenge

Derive a class of arrows from two copies of a class of colored points, sharing the color.

- Structured implementation: provide two methods `from` and `to` that return a view of origin and target as points.
- Flat implementation: provide two methods `from_x` and `to_x` to access `x` in the origin and in the target.

In each case, add a toplevel method `p` that prints the arrow.

A calculus for multiple inheritance

We obviously need hiding and renaming capabilities:

- already useful for single inheritance.
- crucial for multiple inheritance

The calculus of [Riecke-Stone] *privacy via subsumption* is an obvious candidate.

We need small variations and extensions:

+ a merge operation,
+ relocatable methods (so as to implement merging), and
+ virtual methods.

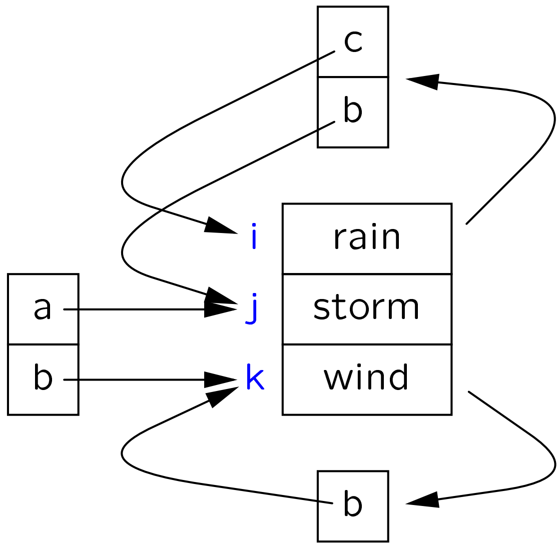
Syntax

$a ::= x \mid \mathtt{fun}\,(x)\,a \mid a\;a$	lambda calculus
$\langle d \rangle_\varphi$	object value
$a \mathbin{@} \varphi$	renaming
$a_1 +_L a_2$	merge of objects
$a.\ell$	access
$d ::= \ell$	virtual method
$\ell \mapsto \varphi, \varsigma(z)\,a$	method with its static dictionary
$\varphi ::=$	finite map from labels to labels
$L ::=$	finite set of labels

+ well-formedness conditions

Relocatable methods

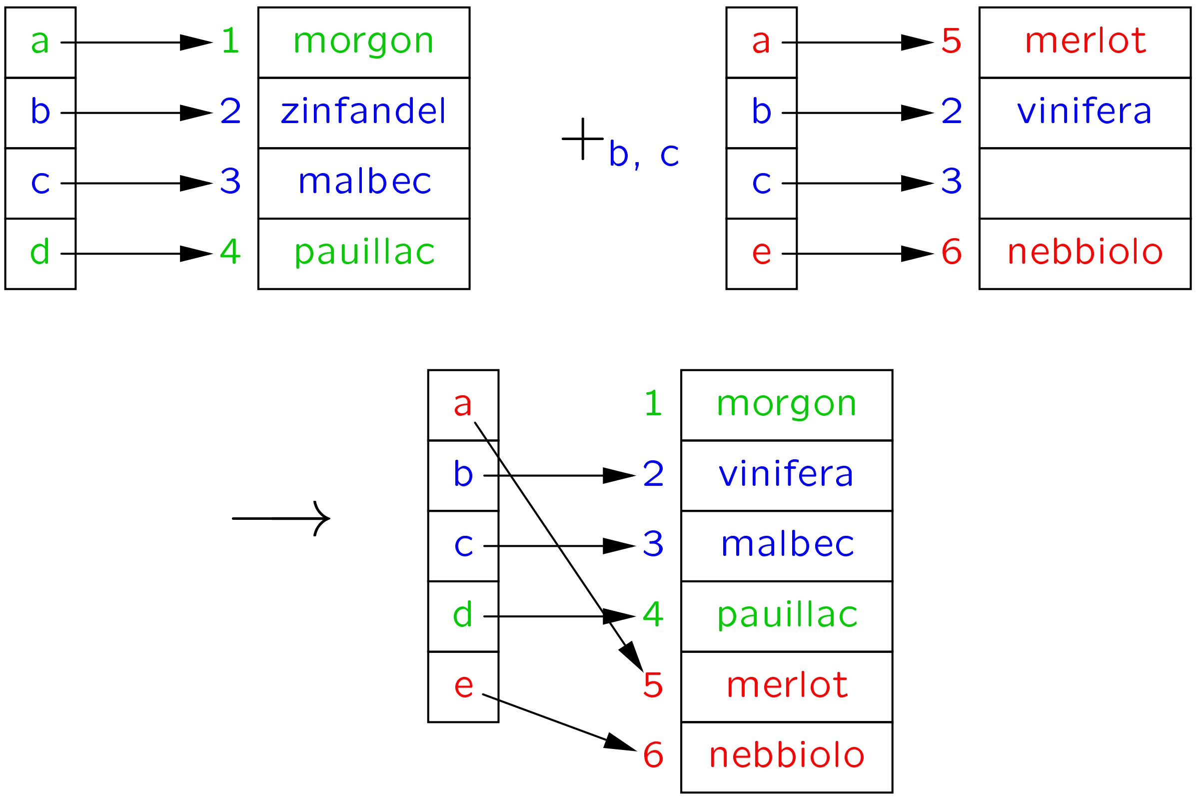
Objects are taken modulo renaming of internal names (slots)



$$d = \begin{pmatrix} i = [c \mapsto i, b \mapsto j], & \text{rain} \\ j = \ldots, & \text{storm} \\ k = [b \mapsto k], & \text{wind} \end{pmatrix}$$

$$\varphi = [a \mapsto j, b \mapsto k]$$

Merging two objects



Formal semantics

$$
\begin{array}{rcll}
\langle d\rangle_\varphi \mathbin{@} \varphi' & \longrightarrow & \langle d\rangle_{\varphi'\circ\varphi} & \\
\langle d\rangle_\varphi.\ell & \longrightarrow & a_i[\langle d\rangle_{\varphi_i}/z] & \text{if } d(\varphi(\ell)) = (\varphi_i, \varsigma(z)\, a_i) \\
\langle d_1\rangle_{\varphi_1} +_L \langle d_2\rangle_{\varphi_2} & \longrightarrow & \langle d\rangle_\varphi & \text{if } (1)
\end{array}
$$

Side conditions (1)

- Merge is defined only if $L \subset \mathit{dom}(\varphi_1) \cap \mathit{dom}(\varphi_2)$.
- d_1 and d_2 may be renamed such that $\mathit{dom}(d_1) \cap \mathit{dom}(d_2) = \varphi_1(L) = \varphi_2(L) = M$.
- Let M_2 be the subset of M that are virtual in d_2 and M_1 be $M \setminus M_2$.
- Then d is $(d_1 \setminus M_2) \cup (d_2 \setminus M_1)$ and φ is $(\varphi_1 \setminus \mathit{dom}(\varphi_2)) \cup \varphi_2$.

Syntactic sugar

Object creation (omitting dictionaries)

$\langle \ell \mapsto \varsigma(z)\, a_\ell, \ell \in L\rangle$ $\quad$ $\langle \ell \mapsto \mathtt{id}_L, \ell \mapsto \varsigma(z)\, a_\ell\rangle_{\mathtt{id}_L}$

Simple inheritance

$a \leftarrow\!\!+_L \langle b\rangle$ $\quad$ $a +_L \langle L, b\rangle$

Allocation

$a + L$ $\quad$ $a +_\emptyset \langle L\rangle$

Projection

$a \,/\, L$ $\quad$ $a \mathbin{@} \mathtt{id}_L$

Extensions

$a - L$	deletes L (becomes virtual)
$a \setminus L$	hides L

Arrows as pair of points

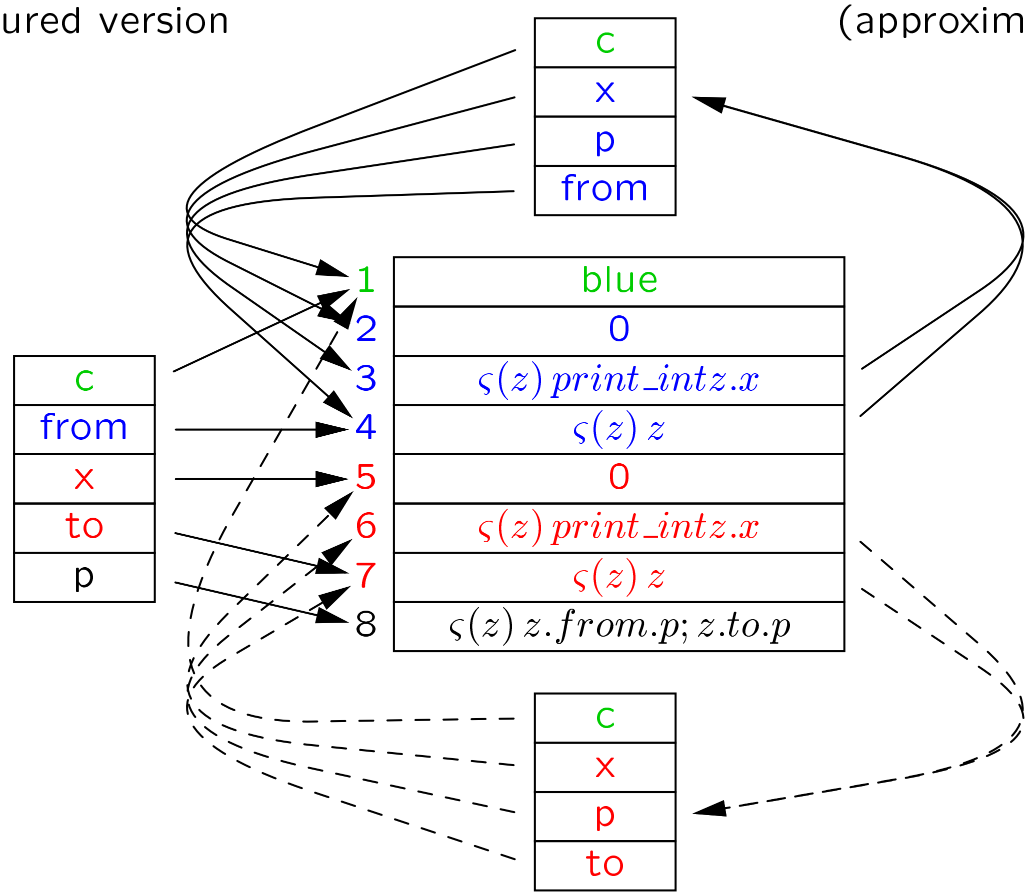
Structured version

$$
\begin{array}{rcl}
\mathit{Point} & \triangleq & \langle c = \mathit{blue}, x = 0, p = \varsigma(z)\, \mathit{print\_int}\ z.x \rangle \\
\mathit{Origin} & \triangleq & \mathit{Point} \leftrightarrow_{c,x,p} \langle \mathit{from} \mapsto \varsigma(z)\, z \rangle \\
\mathit{Target} & \triangleq & \mathit{Point} \leftrightarrow_{c,x,p} \langle \mathit{to} \mapsto \varsigma(z)\, z \rangle \\
\mathit{Arrow} & \triangleq & (\mathit{Origin} +_c \mathit{Target}) \\
 & & \quad \leftrightarrow_{\mathit{from},\mathit{to}} \langle p \mapsto \varsigma(z)\, (z.\mathit{from}.p; z.\mathit{to}.p) \rangle
\end{array}
$$

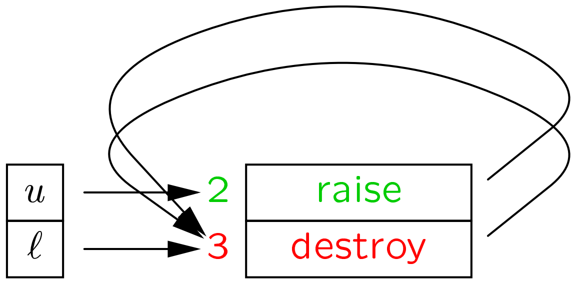
Flat version

$$
\begin{array}{rcl}
\mathit{Origin} & \triangleq & \mathit{Point} \mathbin{@} [c \mapsto c, \mathit{from\_x} \mapsto x, \mathit{from\_p} \mapsto p] \\
\mathit{Target} & \triangleq & \mathit{Point} \mathbin{@} [c \mapsto c, \mathit{to\_x} \mapsto x, \mathit{to\_p} \mapsto p] \\
\mathit{Arrow} & \triangleq & (\mathit{Origin} +_c \mathit{Target}) \\
 & & \quad \leftrightarrow_{\mathit{from\_p},\mathit{to\_p}} \langle p \mapsto \varsigma(z)\, (z.\mathit{from\_p}; z.\mathit{to\_p}) \rangle
\end{array}
$$

Structured version (approximately)

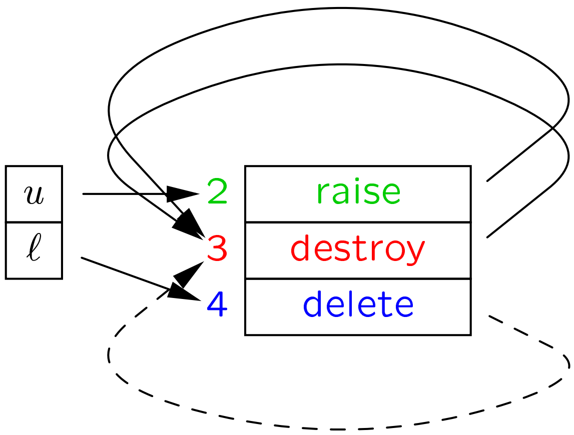


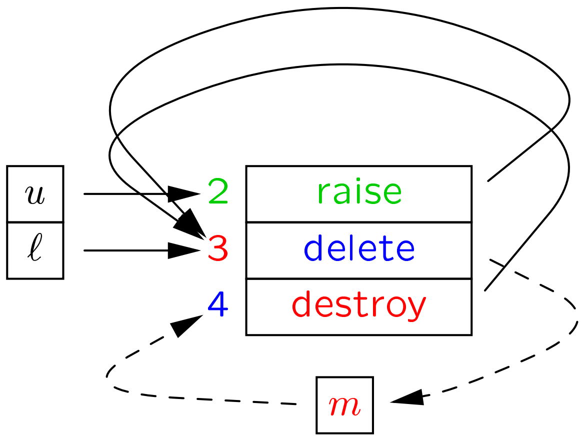
Internalizing super-bindings



Easy
but Incorrect

Correct
but difficult





Method duplication

$$a.m \Leftarrow \ell$$

Semantics

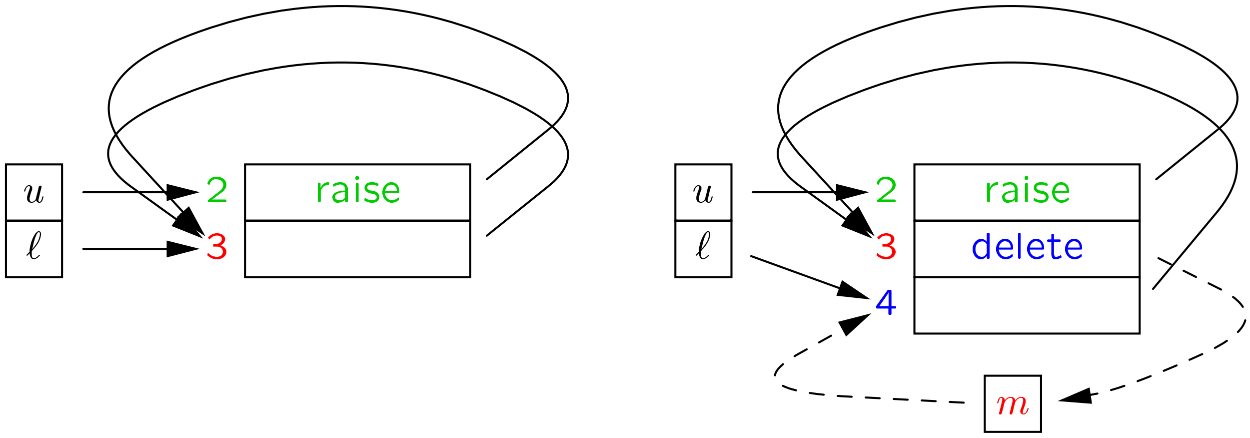
It overrides the definition of m with that of ℓ:

$$\langle d \rangle_\varphi.m \Leftarrow \ell \longrightarrow \langle d \setminus \varphi(m), \varphi(m) \mapsto d(\varphi(\ell)) \rangle_\varphi$$

Example

$$((a + m).m \Leftarrow \ell) \leftrightarrow_L \langle \ell = \varsigma(z) \ \ldots (z.m) \ldots \rangle$$

Super-binding of a virtual method!



$$((a+m) \leftarrow\!\!+_L \langle \ell = \varsigma(z)\ \ldots(z.m)\ldots\rangle) \,@\, [L', m \mapsto \ell]$$

First-order typing rules

Types

$$\tau ::= \quad \alpha \mid \tau \to \tau \mid \langle \ell_i : \tau_i \rangle^{L}$$

L is the set of virtual methods. They must be traced carefully, because they can never be forgotten.

Object

$$\rho = (v : \tau_v, v \in V; c : \tau_c, c \in C)$$

$$\frac{A, z : \langle \rho \circ \varphi_c \rangle^{\emptyset} \vdash a_c : \tau_c \qquad V \subset \mathtt{range}\ (\varphi)}{A \vdash \langle \ell_v; \ell_c = \varphi_c, \varsigma(z)\, a_c \rangle_{\varphi} : \langle \rho \circ \varphi \rangle^{\varphi^{-1}(V)}}$$

Typing merge

Virtual methods can be overridden, but cannot be hidden.

$$\frac{\begin{array}{c} A \vdash a_1 : \langle \tau_1 \rangle^{V_1} \qquad A \vdash a_2 : \langle \tau_2 \rangle^{V_2} \\ C_i = \mathit{dom}\,(\tau_i) \setminus V_i \qquad V = (V_1 \cup V_2) \setminus (C_1 \cup C_2) \\ V_1 \cap \mathit{dom}\,(\tau_2) \subset L \qquad \tau_1 \restriction L = \tau_2 \restriction L \qquad \tau = (\tau_1 \setminus C_2) \cup \tau_2 \end{array}}{A \vdash a_1 +_L a_2 : \langle \tau \rangle^V}$$

Method invokation is possible when no more method is virtual:

$$\frac{A \vdash a : \langle \ell : \tau; \rho \rangle^{\emptyset}}{A \vdash a.\ell : \tau}$$

See also work on virtual methods

[Bono-Bugliesi-Liquori] *Incomplete objects*

[Remy] *From classes to objects via subtyping*.

Subsumption

Methods can be hidden at any time by subsumption.

$$\frac{A \vdash a : \langle \rho \rangle^V \qquad \ell \notin V \qquad V \subset W}{A \vdash a : \langle \rho \setminus \ell \rangle^W}$$

Explicit coercions (instead of subsumption) could maintain a correspondence between the object types and the domains of their dynamic dictionaries. Then, merging could be based on dynamic dictionaries:

$$\langle d_1 \rangle_{\varphi_1} + \langle d_2 \rangle_{\varphi_2} \longrightarrow \langle d_1 \rangle_{\varphi_1} +_{\mathit{dom}\,\varphi_1 \cap \mathit{dom}\,\varphi_2} \langle d_2 \rangle_{\varphi_2}$$

Cost of subsumption: Overridden methods must be listed explicitly.

Stratifying classes and objects

It is common (good) practice to separate classes, which can be inherited, from real objects whose methods can only be invoked or overridden.

This can be done at the level of types, making class types $\langle\rho\rangle^L$ and object types $\langle\rho\rangle$ distinct.

Either subsumption or the `new` primitive could be used to coerce a class of type $\langle\rho\rangle^\emptyset$ into an object of type $\langle\rho\rangle$.

Label groups

Listing overriding labels explicitly during merge is inconvenient.

Bad solution: use new label annotations in classes to indicate which labels should be hidden or overridden.

However, a same label can be used differently in two contexts.

Better: define set of labels as extra class fields, and use these sets (and statically computable operations on them) to describe the set of overridable labels.

$d ::=$	$\dots$	method definitions
	$g = L$	group definition
$L ::=$	$\ell \mid a.g$	groups
	$L_1 \cap L_2 \mid L_1 \cup L_2 \mid L_1 \setminus L_2 \mid \dots$	operation on groups

The arrow example with groups

Overridable labels can be specified as groups:

$$\begin{array}{rcl} Point & \triangleq & \langle all = c, x, p; style = c; \\ & & \quad c = blue, x = 0, p = \varsigma(z)\, print\_int\ z.v \rangle \\ Origin & \triangleq & Point \leftrightarrow_{Point.all} \langle from \mapsto \varsigma(z)\, z \rangle \\ Target & \triangleq & Point \leftrightarrow_{Point.all} \langle to \mapsto \varsigma(z)\, z \rangle \\ Arrow & \triangleq & (Origin +_{Point.style} Target) \end{array}$$

Labels of $Point.style$ (here c) must be visible when used in merging:

$$(Origin \backslash c) +_{Point.style} Target$$

would is incorrect.

Sharing a common antecedent

This implies sharing possibly hidden slots that have the same identity (originating from the same class)

$$\begin{array}{rcl} Point & \triangleq & \langle x = 0, p = \varsigma(z)\, print\_int\ z.v \rangle \\ Color & \triangleq & \langle c = blue \rangle \qquad\qquad Cpoint \triangleq Point + Color \\ Origin & \triangleq & Cpoint \leftrightarrow_{Cpoint.all} \langle from \mapsto \varsigma(z)\, z \rangle / \ from \\ Target & \triangleq & Cpoint \leftrightarrow_{Cpoint.all} \langle to \mapsto \varsigma(z)\, z \rangle / \ to \\ Arrow & \triangleq & (Origin +_{Color} Target) \end{array}$$

This is safe only if the types of shared slots agree.

- This is obviously the case for monomorphic classes.
- For parametric classes, one may need a weak form of dynamic types or a static knowledge of the inheritance hierarchy (could inheritance subtyping as in Moby help?)

Second-order system

The first-order calculus is too weak with functional objects:

- $\varsigma(z)\, z$ returns self with its static view (*i.e.* the dictionary at the time the method is defined) and not its dynamic view.
- Same for functional update.

A second-order calculus must distinguish between the early and dynamic views inside methods. The dynamic view is unknown and must be treated abstractly.

See works on views and renaming

[Riecke-Stone] *Privacy via subsumption*

[Vouillon] recent work (self types + binary method, no M.I.)

Conclusions

To achieve its goal —strong modularity— multiple inheritance must come with renaming capabilities and some way of describing sharing constraints during the merge of two classes.

This can be achieved **reusing** views and adding label groups.

Our exploration is **just an easy first step**:

- There is space left for the design of the surface language: finding the best combination of primitive operators.
- Self types and binary methods need more investigation,
- Parametric classes also need to be treated with care,

Dictionaries or views seem to be unavoidable.

This might be one of the most important progress in our understanding of objects in the last couple of years.

Connection with modules

Mixin modules

- The counterpart to inheritance for modules
- Include merging, renaming, and hiding, but no notion of self types, although... recent work of [Ancona-Zucca] adding methods and a late-binding mechanism to mixin modules.

Does multiple inheritance make the connection easier or harder?

See also

[Duggan-Sourelis 96, Ancona-Zucca 97-99, Flinder-Flatt, Flatt *et al*]

modules $+_{\text{best}}$ classes ?