Lazy mixins and disciplined effects

Keiko Nakata
CNAM/INRIA
Keiko.Nakata@inria.fr

Abstract

Programming languages are expected to support programmer's effort to structure program code. The ML module system, object systems and mixins are good examples of language constructs promoting modular programming. Among the three, mixins can be thought of as a generalization of the two others in the sense that mixins can incorporate features of ML modules and objects with a set of primitive operators with clean semantics. Much work has been devoted to build mixin-based module systems for practical programming languages. In respect of the operational semantics, previous work notably investigated mixin calculi in call-by-name and call-by-value evaluation settings. In this paper we examine a mixin calculus in a call-by-need, or lazy, evaluation setting. We demonstrate how lazy mixins can be interesting in practice with a series of examples, and formalize the operational semantics by adapting Ancona and Zucca's concise formalization of call-by-name mixins. We then extend the semantics with constraints to control the evaluation order of components of mixins in several ways. The main motivation for considering the constraints is to produce side effects in a more explicit order than in a purely lazy, demand-driven setting. We explore the design space of possibly interesting constraints and consider two examples in detail.

1. Introduction

Modularity is an important factor in the development of large programs. In particular programmer's effort to logically organize program code is of great importance in the long run to maintain, debug and extend the program code. Many modern programming languages have mechanisms to support this effort by facilitating modular development of programs. Examples of these mechanisms are object systems and the ML module system.

Being ML programmers, we enjoy the rich expressivity of the ML module system for modular programming. Nestable structures allow us to hierarchically organize namespaces and program code. With signature constraints, we control visibility of components of structures. In particular the combination of nesting and signature constraints offers fine grained visibility control, as witnessed, for example, by the Moby programming language (Fisher and Reppy 1999). Functors, which are functions on modules, facilitate code reuse in a modular way. Although functors might not be as pervasive as nesting or signature constraints in our programs, they play a critical role in some contexts. Good examples are Map.Make and Set.Make functors as implemented in OCaml's standard library.

However ML does not have recursive modules. That is, neither recursive functions nor types can be defined across module boundaries. As a result of this constraint, programmers may have to consolidate conceptually separate components into a single module, intruding on modular programming (Russo 2001). Compared to ML, object systems have excellent support for recursion across object/class boundaries. Particularly in the presence of late-binding, object systems facilitate the development of extensible programs, where recursively defined types and functions may need to be extended together. Typical such scenarios are condensed in the notorious expression problem (Torgersen 2004).

We expect a module system to support all the familiar features of ML modules as well as recursion between modules and late-binding. There are at least three approaches to design such a module system. Two of them are to extend the ML module system with recursion (Crary et al. 1999) and to extend object systems with nesting, abbreviation and type members (Odersky et al. 2003). The third approach is to develop another form of a module system, namely mixins. In this paper we follow this third approach.

The concept of mixins is first introduced in the context of object systems (Bracha 1992), then is extended to in the context of ML-style modules. A mixin is a collection of named components, where each component can be either defined (bound to a definition) or deferred (declared without definition). Two key operations on mixins are the sum and freeze operations; the former takes two mixins and composes a new mixin by merging the two, and the latter resolves, or links, deferred components of a mixin to defined ones. The sum operation is reminiscent of functor application in ML and inheritance in object systems. However more flexibility is obtained by separating the resolution from the sum operation. The freeze operation is free to resolve a deferred component to a defined one independently of their names at any point, as long as their types match; notably it can liberally tie a recursive knot inside a mixin. Indeed mixins are designed to be a generalization of ML-style modules and objects, by incorporating features of both with a set of primitive operators with clean semantics (Bracha 1992; Ancona and Zucca 2002).

Two challenging problems remain to replace ML-style modules with mixins, namely type checking and initialization. In this paper, we address the latter problem. Promising progress has been made in designing a type system for mixins with type components and a signature language to enforce type abstraction between mixins (Odersky et al. 2003; Owens and Flatt 2006); we expect to benefit from the previous work for the former problem.

Initialization of mixins poses an important design problem if we are to build a mixin-based module system on top of a call-by-value core language supporting arbitrary side-effects, such as the ML core language. The main difficulty stems from unconstrained recursive definitions such as $\mathsf{let\ rec\ x = x}$ that the core language does not allow but that might result from the freeze operation. We

need an initialization semantics for mixins which takes account of unconstrained recursion consistently with the call-by-value semantics of the core language; this is the subject of this paper.

Ancona and Zucca's mixin calculus (Ancona and Zucca 2002), called *CMS*, is one of the most influential work in formalizing an operational semantics for mixins. The formalization is elegantly concise and the paper illustrates how mixins support various existing constructs for composing modules, found in the ML module system, object-systems and linking calculi (Cardelli 1997), well-explaining why we claim mixins are a generalization of other modular systems. Inspired by *CMS*, Hirschowitz and Leroy examined a mixin calculus in a call-by-value setting to build a mixin-based module system on top of the ML core language. The original call-by-name semantics of *CMS* might not be well-suited for the call-by-value effectful ML core language; *CMS* admits recursive definitions such as `let rec x = x`, causing the evaluation to diverge when `x` is selected, and it can produce the same side-effect repeatedly. Neither behavior is consistent with the semantics of ML.

In this paper we examine a mixin calculus in a call-by-need, or lazy, evaluation setting with a back-patching semantics. Broadly we model mixins as nestable records with lazy fields. In the simplest setting which we examine in Section 3, a component of a mixin is evaluated when it is selected. We explain how side-effects are duplicated and linearly produced by *open* and *closed* mixins respectively, while allowing both open and closed mixins to be merged indiscriminately; this tolerance of merging mixins of different status is our important design choice, which is different from previous proposals (Hirschowitz and Leroy 2005; Ancona et al. 2003). In Section 2.1, we exemplify how the tolerance can be useful.

Then in Section 4, we extend the former semantics of Section 3 with an ability to constrain the evaluation order of components of mixins. While the former semantics admits the most flexible recursive initialization patterns for mixins, the flexibility can do harm because of the intrinsically implicit evaluation order determined at run-time in a demand-driven way. It may be interesting to constrain possible initialization patterns, making the evaluation order more explicit. For instance, we may want to enforce top-down evaluation order within mixins, where components of a mixin are evaluated following the textual definition order in the source program. Or, we may want to keep the invariant that once a component of a mixin is selected, all its components are eventually evaluated. Indeed our ultimate goal is to find the most beneficial constraint on the evaluation order that still admits interesting recursive initialization patterns, but that makes the evaluation order more explicit, thus more predictable to programmers.

In this paper we do not propose a particular constraint; simply we do not yet have enough experience in programming with mixins to decide what constraint is best in practice. Hence we formalize the operational semantics so that it deals with several constraints. Concretely, we give a constraint language and extend the semantics of Section 3 to evaluate components of mixins according to a given constraint expressed by the constraint language. We explore the design space of possibly interesting constraints and examine two particular constraints in detail as examples.

Contributions of the paper are summarized as follows. We formalize the operational semantics for a lazy mixin calculus (Section 3), by moderately extending Ancona and Zucca's formalization of a call-by-name mixin calculus (Ancona and Zucca 2002). We demonstrate how lazy mixins can be useful in practice through examples (Section 2). Then we extend the semantics to be able to control the evaluation order of components of mixins in several ways (Section 4) and exemplify concrete scenarios where particular evaluation strategies are enforced by constraints. We believe the ability to deal with several evaluation strategies is a novelty of the formalization and the formalization serves as a basis for exploring the design space.

```
module MakeSet = functor (X : sig
  val create : unit → int val compare : int → int → int end) →
  struct
    let create () = [ X.create () ]
    let compare s1 s2 = ..... X.compare ...
  end
module MakeMultiSet = functor(X : sig
  val create : unit → int val compare : int → int → int end) →
  struct
    let create () = [[ X.create () ]]
    let compare s1 s2 = .... X.compare ...
  end
module Key = struct
  let count = ref (-1)
  let create () = incr count; !count
  let compare x y =
    if x = y then 0 else if x < y then 1 else (-1)
end
module Set = MakeSet(Key)
module MultiSet = MakeMultiSet(Key)
```

Figure 1. MakeSet and MakeMultiSet ML functors

2. Examples

In this section, we introduce our lazy mixin calculus through a series of examples. Many constructs of the calculus come from *CMS* (Ancona and Zucca 2002). Examples are written in a more programmer-friendly surface syntax and we assume a small subset of the OCaml core language for the core language of the mixin calculus. We recall that the OCaml core language adopts a call-by-value evaluation strategy and supports arbitrary side-effects within (core) expressions.

2.1 MakeSet and MakeMultiSet mixins

We start by looking at possible mixin equivalents to MakeSet and MakeMultiSet ML functors and their instances by the Key structure, as given in Figure 1. MakeSet and MakeMultiSet are functions on ML structures, or functors, for making sets and multi-sets of integers, internally represented as lists and lists of lists respectively. Both take as argument a structure containing a create function for producing integers out of nothing and a compare function for comparing given two integers. MakeSet extends this functionality to sets of integers and MakeMultiSet does for multi-sets. Then we apply the two functors to the Key structure to instantiate customized Set and MultiSet structures.

Below is a mixin equivalent to the MakeSet functor:

```
mixin MakeSet = {
  val create_element : unit → int
  val compare_element : int → int → int
  let create () = [ create_element () ]
  let compare s1 s2 = ..... compare_element ...    }
```

A *mixin structure* is a sequence of named components, where a component may be defined like create and compare or deferred like create_element and compare_element. The bodies of defined components can refer to names of deferred components as if they were present. Similarly, below is a mixin equivalent to the MakeMultiSet functor:

```
mixin MakeMultiSet = {
  val create_element : unit → int
  val compare_element : int → int → int
  let create () = [[ create_element () ]]
  let compare s1 s2 = ..... compare_element ...    }
```

We build a mixin equivalent to the Key structure in two steps:

```
mixin FKey = {
  let count = ref (-1)
  let create_key () = incr count; !count
  let compare_key x y =
    if x = y then 0 else if x < y then 1 else (-1)   }
mixin Key = close(FKey)
```

We distinguish two states of mixins: open and closed. Intuitively a closed mixin is a record with lazy fields, whereas an open mixin is a function which returns a record with lazy fields. Intended implications of this comparison are 1) projection of components is only possible from closed mixins, but not from open mixins; 2) an open mixin can be instantiated to create closed mixins; 3) side-effects contained in a closed mixin are produced exactly once, whereas side-effects in an open mixin are produced as many times as the mixin is instantiated. FKey above is an open mixin, and Key is a closed mixin instantiated from FKey by the *close* operation. A projection Key.create_key is legal, but FKey.create_key is not. As expected, the counter Key.count is initialized to -1 exactly once.

We may close the FKey mixin again:

```
mixin Key2 = close(FKey)
```

Counters Key.count and Key2.count are distinct from each other. For instance, calling Key.create does not increase Key2.count.

The close operation is only available for mixins without holes, or mixins which do not contain deferred components. The *freeze* operation, possibly combined with the *sum* operation, is used to fill in the holes. For instance, below we merge Key and MakeSet mixins by the sum operation, then resolve the deferred components create_element and compare_element of MakeSet to the defined component create_key and compare_key of Key respectively by the freeze operation, to fill in the holes of MakeSet:

```
mixin FSet' = Key ← MakeSet
mixin FSet = freeze_ψ(FSet')
```

where ψ is the following mapping[1]:

```
⟨ create_element ↦ create_key;
  compare_element ↦ compare_key ⟩
```

Now FSet does not contain holes, hence we can close it:

```
mixin Set = close(FSet)
```

We do the same for MakeMultiSet in one step this time:

```
mixin MultiSet = close(freeze_ψ(Key ← MakeMultiSet))
```

Set and MultiSet mixins are equivalent to Set and MultiSet ML structures as given in Figure 1.

It is important to see differences resulting from closing FKey after merging it with MakeSet and MakeMultiSet as follows:

```
mixin Set' = close(freeze_ψ(FKey ← MakeSet))
mixin MultiSet' = close(freeze_ψ(FKey ← MakeMultiSet))
```

Here FKey is instantiated twice. While Set and MultiSet share the same counter, Set' and MultiSet' have distinct counter's. As a result, calling Set.create affects both results of the next calls to Set.create and MultiSet.create, while calling Set'.create only does the result of the next call to itself.

[1] Only in this section, we use angle brackets instead of square brackets to denote mappings to avoid confusion with list expressions.

2.2 Widget mixins

Many GUI programming APIs involve recursive initialization patterns to form mutually referential graphs among related widgets in the style of *create-and-configure*, as Syme calls in (Syme 2005). In that paper he explains how recursive initialization patterns are prominent in GUI programming and proposes a semi-safe lazy evaluation strategy for the ML core language to tame ML's value recursion restriction; the restriction constrains right-hand sides of recursive definitions to be syntactic values, thus may hinder uses of sophisticated GUI programming APIs. In this subsection, we consider a GUI example similar to his in a lazy mixin setting.

We assume given the following interface of the API:

```
type form type formMenu type menuItem
val createForm: string → form
val createMenu: string → formMenu
val createMenuItem: string → menuItem
val toggle : menuItem → unit
val setMenus : form * formMenu list → unit
val setMenuItems : formMenu * menuItem list → unit
val setAction : menuItem * (unit → unit) → unit
```

The API requires the create-and-configure initialization pattern where widgets are first created, then explicit mutation configures a relation between the widgets. Below we build boiler-plate mixins which encapsulate the create-and-configure pattern:

```
mixin Form = {
  val name : string
  val menus : formMenu list
  let form = createForm(name)
  let _ = setMenus(form, menus)  }
mixin Menu = {
  val name : string
  val items : menuItem list
  let menu = createMenu(name)
  let _ = setMenuItems(menu, items)  }
mixin MenuItem = {
  val name : string
  val other : menuItem
  let item = createMenuItem(name)
  let _ = setAction(item, fun () → toggle(other))  }
```

Then we use them:

```
MyForm =
  hide_name(freeze_⟨name↦name⟩(Form ← { let name = "Form" }))
MyMenu =
  hide_name(freeze_⟨name↦name⟩(Menu ← { let name = "Menu" }))
MyItem1 =
  rename_(⟨other↦item2⟩,⟨item1↦item⟩)(hide_name(
    freeze_⟨name↦name⟩(MenuItem ← { let name = "Rice" })))
MyItem2 =
  rename_(⟨other↦item1⟩,⟨item2↦item⟩)(hide_name(
    freeze_⟨name↦name⟩(MenuItem ← { let name = "Grape" })))
MyGUI =
  close(freeze_ψ1(
    MyItem1 ← (MyItem2 ←(MyMenu ←MyForm))))
```

where ψ_1 is the following mapping:

```
⟨ item1 ↦ item1; item2 ↦ item2;
  items ↦ [item1; item2] ; menus ↦ [menu]⟩
```

Above we have introduced two new constructs. The *hide* operation $\mathsf{hide}_X(M)$ hides the component named X of the mixin M by making the component invisible outside. The *rename* operation $\mathsf{rename}_{(\phi_1,\phi_2)}(M)$ changes names of deferred and defined components of the mixin M by ϕ_1 and ϕ_2 respectively, where ϕ_1 and

ϕ_2 are finite mappings on names of components[2]. For instance, in the definition of MyItem1, the deferred component other is renamed to item2 and the defined component item to item1. Observe the opposite directions of mappings $\langle \mathsf{other} \mapsto \mathsf{item2} \rangle$ for deferred components and $\langle \mathsf{item1} \mapsto \mathsf{item} \rangle$ for defined ones. This adds flexibility of the rename operation in that when ϕ_1 maps deferred components of distinct names to the same name, then the components can be resolved simultaneously, and that when ϕ_2 maps two distinct names X_1 and X_2 to a single name X, then the component which was named X can now projected by either X_1 or X_2. Thanks to the renaming, MyItem1 and MyItem2 can be merged to form MyGUI without causing name clash; a deferred component and a defined component of a mixin can have the same name, but a deferred (resp. defined) component must not have the same name as other deferred (resp. defined) components. Besides, we have used the freeze operation in a more flexible way by mapping a name of a deferred component to an expression composed of names of defined components, for example [item1; item2].

The above example builds a GUI application forming a widget containment hierarchy where MyForm contains MyMenu, which contains MyItem1 and MyItem2. MyItem1 and MyItem2 are mutually recursive; each toggles the activation state of the other. We hide the name component of mixins to be merged to avoid name clash. Anonymous (under-scored) components do not contribute to name clash; they would be implemented as syntax sugar via the hide operation. Renaming of other and item components of MyItem1 and MyItem2, both of which are derived from the same mixin MenuItem, is necessary for cross-connecting the deferred component of one to the defined component of the other. In the last line we sum up the constituent mixins and resolve deferred components to defined ones, configuring the widget containment hierarchy.

This example suggests it can be useful to control the evaluation order of components of mixins, instead of evaluating them purely lazily in a demand-driven way, i.e. evaluating only the components that are projected. Indeed we would like to make sure the widget containment hierarchy has been properly configured before MyGUI is actively used. For that purpose, all the anonymous components of MyGUI must be evaluated before any of its components becomes externally accessible. In Section 4, we present the operational semantics which can enforce such constraint on the evaluation order.

2.3 A combinator library for marshallers

The last example deals with marshaller combinators and is motivated by Syme's paper again (Syme 2005). Kennedy introduced a functional-language combinator library for building marshallers and unmarshallers of data structures (Kennedy 2004). The essential ingredient of his proposal is the tying together of a marshaller/unmarshaller pair in a single value. Then the consistency of marshalling and unmarshalling is ensured by construction. The original proposal of Kennedy is implemented in Haskell. Porting the code to ML is mostly easy, except for a couple of wrinkles. The value recursion restriction of ML is a source for the wrinkles and requires cumbersome workaround for an ML version of the combinator. In this subsection, we rebuild a combinator library for marshallers using lazy mixins with ML as the core language and demonstrate how the use of lazy mixins avoids the value recursion problem. In the following examples we assume a richer mixin language where a mixin may contain deferred and defined types as components, although our formal development does not consider mixins with type components. Type checking of mixins is not in the scope of the paper. As far as our examples are concerned, existing type systems are sufficient (Flatt and Felleisen 1998).

We consider a mixin-based combinator library with the specification given below. A mixin signature mixin M : { type t val x : t } → { type s = t ∗ t val y : s } specifies an open mixin with deferred type component t and value component x of type t, written in the left-hand side of the arrow, and with defined type component s satisfying type equation s = t ∗ t and value component y of type s, written in the right-hand side[3]. The scope of type names declared in the left hand of the arrow extends to the right hand.

```
type channel
type α marshaller
val marshal : α marshaller → α ∗ channel → unit
val unmarshal : α marshaller → channel → α
mixin PairMrshl :
 { type s1 type s2
   val mrshl1 : s1 marshaller  val mrshl2 : s2 marshaller } →
 { type t = s1 ∗ s2  val marshaller : t mrshl }
mixin ListMrshl :
 { type elm  val mrshl_elm : elm marshaller } →
 { type t = elm list  val mrshl : t marshaller }
mixin InnerMrshl :
{ type src  type trg  val f : src → trg  val g : trg → src
  val mrshl_src : src marshaller } →
{ type t = trg  val mrshl : t marshaller }
mixin IntMrsh :
 {} → { type t = int  val mrshl : t marshaller }
mixin StringMrsh :
 {} → { type t = string  val mrshl : t marshaller }
```

IntMrsh and StringMrsh mixin have an empty deferred component. But they are still open mixins, thus needs to be closed for making their components accessible.

The abstract type marshaller could be internally implemented as a record consisting of a marshalling action and unmarshalling action:

```
type α marshaller =   { marshal: α ∗ channel → unit;
                        unmarshal : channel → α }
```

Recall that it is important for consistently building marshallers and unmarshallers that a marshaller is a single value, but not two separate functions. In this way, users of the library can only build consistent marshaller/unmarshaller pairs.

We do not present further details of how the library can be implemented. In the original paper (Kennedy 2004), Kennedy explains an excellent implementation which lets the programmer control sharing of the marshaled data. Transposing marshaller combinators for constructed types such as pairs and lists, originally implemented as functions, to mixins is straightforward. For instance, a function-based marshaller combinator pairMrshl(mrshl1, mrshl2) of type (s1 marshaller) ∗ (s2 marshaller) → (s1 ∗ s2) marshaller for constructing a marshaller for a pair from marshallers of the components can be translated into a mixin as follows:

```
mixin PairMrshl = {
 type s1 type s2
 val mrshl1 : s1 marshaller val mrshl2 : s2 marshaller
 type t = s1 ∗ s2
 let mrshl = (∗ the body of pairMrshl(mrshl1, mrshl2) ∗) }
```

Now we turn to how to build custom-marshallers for user-defined data types. We first build marshallers for both a single file, represented as a pair of an integer and string, and a list of files:

[2] In the formalization, we will distinguish α-convertible *identifiers* (internal names) and non-convertible *names* (external names). Then ϕ_1 and ϕ_2 are mappings on names, not on identifiers.

[3] This signature language is designed only for the sake of the examples. A more practical signature language is proposed, for instance in (Owens and Flatt 2006).

```
type file = int * string
mixin FileMrshl = freeze*(
  (rename_(∅,⟨s1↦t;mrshl1↦mrshl⟩)(IntMrshl)) ←
  (rename_(∅,⟨s2↦t;mrshl2↦mrshl⟩)(StringMrshl)) ← PairMrshl))
mixin FilesMrshl = freeze*(
  (rename_(∅,⟨elm↦t;mrshl_elm↦mrshl⟩)(FileMrshl)) ← ListMrshl)
```

The notation $\emptyset$ denotes an empty mapping. Above we have introduced a high-level mixin construct freeze*, which resolves deferred components to the same-named defined components if exists, then hides the defined components used. The formalization given in the next section does not include freeze*. For the surface language, we could implement it by combining freeze and hide operations with the help of the type system.

Next we build marshallers for both a single folder and a list of folders, which form recursive data structures:

```
  type folder = { files: file list; subfldrs: folders }
  and folders = folder list
```

Like Syme, we use an intermediate mixin in favor of conciseness.

```
mixin FldrInnerMrshl = freeze*(
  { type src = folder type trg = folders
    let f (fls, fldrs) = { files = fls; subfldrs = fldrs }
    let g fld = (fld.files, fld.subfldrs) } ←
  InnerMrshl)
```

Then to build marshallers for folder and folders, we follow their type definitions, by merging constituent mixins and resolving deferred components to defined ones:

```
mixin FldrMrshl = close(freeze*(
  (rename_(∅,⟨s1↦t;mrshl1↦mrshl⟩)(FilesMrshl)) ←
  (hide_t(rename_(⟨s1↦s1;s2↦s2;mrshl1↦mrshl1;mrshl2↦mrshl2⟩,
                  ⟨mrshl_src↦mrshl⟩)(PairMrshl))) ←
  (rename_(⟨mrshl_src↦mrshl_src⟩,⟨elm↦t;mrshl_elm↦mrshl;fldMrshl↦mrshl⟩)
          (FldrInnerMrshl)) ←
  (rename_(⟨elm↦elm;mrshl_elm↦mrshl_elm⟩,
           ⟨fldrsMrshl↦mrshl;mrshl2↦mrshl;s2↦t⟩)(ListMrshl))
```

Thus we have created marshaller FldrMrshl.fldrMrshl for a single folder and FldrMrshl.fldrsMrshl for a list of folders. In the surface language we could provide an identity mapping with appropriate domain to improve notational verbosity. For the above example, we preferred not to use identity mappings to make explicit how deferred components are resolved to defined ones.

Specifying marshallers for recursive data types such as folder and folders is not problematic in the original Haskell context of Kennedy or in our lazy mixin context, essentially due to the laziness. But it is problematic in the context of the ML core language because of the value recursion restriction. The problem is well-discussed in the previous papers of Kennedy and Syme. In short, it is due to ML's intolerance of the following recursive definition[4]

```
let rec p = pairMrshl(p, intMrshl)
```

where we assumed primitive marshaller intMrshl and marshaller combinator pairMrshl given in the library. Note that the abstraction of type marshaller is not the root of the problem. Even we exposed the underlying implementation of marshaller, we still face the problem since marshallers are *pairs* of functions.

If we consider first-class mixins, i.e. mixins as core values, we could implement a marshaller as separate functions of marshal action and unmarshal action, while ensuring the consistency of constructed marshaller/unmarshaller pairs. Our formalization of a lazy mixin calculus is largely abstracted over the core language. The core language may support first-class mixins, but we preferred the current presentation in favor of generality by not assuming a richer core language. We also found the current presentation useful to demonstrate a possibly interesting scenario where lazy mixins and a call-by-value core language are combined to obtain more tolerant recursive definitions.

[4] The example is not ideal in that ML does not allow recursive type definitions such as `type t = t * int`. But the source of the problem should be clear.

x, y	$\in$	*Idents*	*identifiers*
X, Y	$\in$	*Names*	*names*
l	$\in$	*Loc*	*locations*
E	::=	$C \mid M$	*expressions*
M	::=	$[\iota; o; \rho]$	*mixin structure*
	$\mid$	$M_1 \leftarrow M_2$	*sum*
	$\mid$	$\phi_1 \triangleleft M \triangleright \phi_2$	*rename*
	$\mid$	$hide_X(M)$	*hide*
	$\mid$	$freeze_\psi(M)$	*freeze*
	$\mid$	$close(M)$	*close*
	$\mid$	$M.X$	*projection*
	$\mid$	x	
	$\mid$	X	
	$\mid$	l	*location*
C	::=	$M.X \mid l$	*projection, location*
	$\mid$	$x \mid X$	*identifier, name*
	$\mid$	$\ldots$	
ι	::=	$x_i \overset{i\in I}{\mapsto} X_i$	*input assignment*
o	::=	$X_i \overset{i\in I}{\mapsto} x_i$	*output assignment*
ρ	::=	$x_i \overset{i\in I}{\mapsto} E_i$	*local binding*
ϕ	::=	$X_i \overset{i\in I}{\mapsto} Y_i$	*renaming*
ψ	::=	$X_i \overset{i\in I}{\mapsto} E_i$	*tying*

Figure 2. Syntax for *Lyre*

3. Lazy mixins

We start by considering a simpler semantics, where a component of a mixin is evaluated when it is projected. The syntax of our lazy mixin calculus, named *Lyre*, is defined in Figure 2. We assume pairwise disjoint sets *Idents* of identifiers, *Names* of names, and *Loc* of locations. Components of a mixin are internally referred to by (α-convertible) identifiers, but externally accessed by (non-convertible) names. We use locations to formalize lazy evaluation.

Notations For a finite mapping f, $dom(f)$ and $ran(f)$ respectively denote the domain and range of f. $\emptyset$ is an empty mapping, that is, $dom(\emptyset)$, $cod(\emptyset)$ and $ran(\emptyset)$ are empty sets. The notation $a_i \overset{i\in I}{\mapsto} b_i$ denotes the finite mapping f such that, for all $i \in I$, $f(a_i) = b_i$. It is only defined when, for all $i, j \in I$, $i \neq j$ implies $a_i \neq a_j$. Throughout the paper, we only consider finite mappings, so simply say a mapping to mean a finite one.

For a mapping f, $f\backslash x$ is the restriction of f to $dom(f)\backslash\{x\}$.

For mappings f, f', we write $f + f'$ for the union of f and f'. That is, $dom(f + f') = dom(f) \cup dom(f')$, and for all x in $dom(f + f')$,

$$(f + f')(x) = \begin{cases} f(x) & \text{when } x \in dom(f) \\ f'(x) & \text{when } x \in dom(f') \end{cases}$$

$f + f'$ is defined only if $dom(f) \cap dom(f') = \emptyset$. The notation $f' \circ f$ denotes the mapping composition. It is defined only when $ran(f) \subseteq dom(f')$.

We have explained most of the constructs for mixin expressions in the previous section. In the formalization, however, mixin structures take a more fundamental form. Precisely, a mixin structure, simply called structure hereafter, is a triple of *input assignment* ι, *output assignment* o and *local binding* ρ. The local binding ρ is

a mapping from identifiers to expressions and corresponds to the body of the mixin; if x is in $dom(\rho)$, then x is a defined component of the mixin with $\rho(x)$ being the defining expression. $\rho(x)$ can refer to both defined and deferred components of the mixin via identifiers. Any identifier in $dom(\rho)$ and $dom(\iota)$ is bound in $\rho(x)$. $\rho(x)$ must not contain names. The input assignment ι is a mapping from identifiers to names and corresponds to declarations of deferred components; a deferred component internally referred to by x is resolved by the name $\iota(x)$. $dom(\rho)$ and $dom(\iota)$ must be disjoint. The output assignment o is a mapping from names to identifiers; a component of a mixin externally accessed by X is associated to $o(X)$ inside the mixin. $ran(o)$ must be a subset of $dom(\rho) \cup dom(\iota)$. Structures are identified up to α-renaming of identifiers. The explicit distinction between identifiers and names allows identifiers to be renamed by α-conversion, while names remain immutable, thus making projection by name unambiguous (Hirschowitz and Leroy 2005; Lillibridge and Harper 1994).

In agreement with the distinction of identifiers and names, the rename operation takes two mappings on names, and the hide operation takes a name as argument. The freeze operation takes a mapping from names to expressions, where expressions may contain names. Mixin expressions contain locations. We use locations in the operational semantics to implement lazy evaluation, but locations will not appear in the surface language.

The formalization is mostly independent of the core language. We only assume the core language includes projection from mixin expressions, locations, identifiers and names. Again locations will not appear in the surface language.

In Figure 3, we define the operational semantics for *Lyre*. A *heap state* σ is a mapping from locations to *heap objects*, which are either expressions, values or `error`. We formalize lazy evaluation by suspending and memorizing evaluation in heap states. We assume given core values v. Then values V are either structures or core values. We syntactically distinguish structures as mixin expressions, surrounded by square brackets, and as values, surrounded by angle brackets. The distinction lets us simplify the formalization.

The judgment $\sigma \vdash E \downarrow (V; \sigma_2)$ means that in heap state σ expression E evaluates into value V with heap state being σ_2. We assume given inference rules to deduce $\sigma \vdash C \downarrow (V; \sigma_2)$ for core expressions other than projections or locations.

Notations We write $\sigma[l_i \overset{i \in I}{\mapsto} \kappa_i]$ to denote a mapping extension. Precisely,

$$\sigma[l_i \overset{i \in I}{\mapsto} \kappa_i](l') = \begin{cases} \kappa_i & \text{when } l' = l_i \text{ for some } i \in I \\ \sigma(l') & \text{otherwise} \end{cases}$$

The notation is defined only if, for any $i, j \in I$, $i \neq j$ implies $l_i \neq l_j$. We may write $\sigma[l \mapsto \kappa]$ when I is a singleton. The notation $E[l_i/x_i]_{i \in I}$ denotes the substitution of l_i's for x_i's in E for all $i \in I$. The notation is defined only if, for any $i, j \in I$, $i \neq j$ implies $x_i \neq x_j$. For input assignment ι, tying ψ and output assignment $o = X_i \overset{i \in I}{\mapsto} x_i$, $o \circ \psi \circ \iota$ is the mapping ρ such that $dom(\rho) = dom(\iota)$ and, for all $x \in dom(\iota)$, $\rho(x)$ is the expression obtained from $\psi \circ \iota(x)$ by replacing X_i's with x_i's for all $i \in I$. $o \circ \psi \circ \iota$ is only defined when $ran(\iota) \subseteq dom(\psi)$ and, for all $x \in dom(\iota)$, any name appearing in $\psi \circ \iota(x)$ is in $dom(o)$.

Let's look at the inference rules. A structure evaluates into itself and the heap state is unchanged (rule (1)). The sum operation merges the two operand mixins (rule (2)). The side conditions ensure that identifiers do not collide and are always satisfiable by taking appropriate α-equivalent mixins. The effect of the rename operation is simply compositions of mappings (rule (3)). The hide operation narrows the domain of the output assignment (rule (4)). The freeze operation resolves deferred components according to the tying ψ (rule (5)). Precisely, for all $x \in dom(\iota_1)$, x is resolved to the expression $o \circ \psi \circ \iota_1(x)$. $\psi \circ \iota_1(x)$ may contain names, which are replaced with identifiers by o. The rule augments the local binding with $o \circ \psi \circ \iota_1$ and the input assignment of the resulting mixin diminishes accordingly.

The rule (6) for the close operation is responsible for making mixins lazy. Firstly the operand mixin expression must be evaluated into a structure without holes. Then, for each defined identifier x_i, a fresh heap location l_i is allocated to store the defining expression E_i, where any occurrence of x_j's in E_i is substituted by l_j's. The body of the resulting structure maps x_i to l_i, thus access to x_i is redirected to l_i.

To evaluate projection $M.X$ (rule (7)), M is first evaluated into a structure $\langle \iota; o; \rho \rangle$. Then the rule consults o for the associated identifier to X, thus determines the expression that $M.X$ accesses by looking up the identifier in ρ. In the normal evaluation, i.e. when projection is made from a closed mixin, $\rho \circ o(X)$ returns a location, where the defining expression of $o(X)$ is stored.

The last two rules are for evaluating locations l, and are fairly standard. If the heap state contains a value at l (rule (8)), then the value is returned and the heap state is unchanged. When an expression E is stored at l, then it is evaluated. The heap state maps l to `error` during the evaluation of E, to avoid evaluating the same

$$\begin{array}{lcll} V & ::= & \langle \iota; o; \rho \rangle \mid v & \textit{values} \\ \kappa & ::= & E \mid \texttt{error} \mid V & \textit{heap objects} \\ \sigma & \in & Loc \text{ -fin-> } \kappa & \textit{heap state} \end{array}$$

$$\textit{mixin structure} \quad \frac{}{\sigma \vdash [\iota; o; \rho] \downarrow (\langle \iota; o; \rho \rangle; \sigma)} \quad (1)$$

$$\textit{sum} \quad \frac{\sigma \vdash M_1 \downarrow (\langle \iota_1; o_1; \rho_1 \rangle; \sigma_2) \quad \sigma_2 \vdash M_2 \downarrow (\langle \iota_2; o_2; \rho_2 \rangle; \sigma_3) \quad dom(\rho_1) \cap dom(\rho_2) = \emptyset \quad dom(\iota_1) \cap dom(\iota_2) = \emptyset}{\sigma \vdash M_1 \leftarrow M_2 \downarrow (\langle \iota_1 + \iota_2; o_1 + o_2; \rho_1 + \rho_2 \rangle; \sigma_3)} \quad (2)$$

$$\textit{rename} \quad \frac{\sigma \vdash M \downarrow (\langle \iota; o; \rho \rangle; \sigma_2)}{\sigma \vdash \phi \lhd M \rhd \phi' \downarrow (\langle \phi \circ \iota; o \circ \phi'; \rho \rangle; \sigma_2)} \quad (3)$$

$$\textit{hide} \quad \frac{\sigma \vdash M \downarrow (\langle \iota; o; \rho \rangle; \sigma_2)}{\sigma \vdash hide_X(M) \downarrow (\langle \iota; o \backslash X; \rho \rangle; \sigma_2)} \quad (4)$$

$$\textit{freeze} \quad \frac{\sigma \vdash M \downarrow (\langle \iota_1 + \iota_2; o; \rho \rangle; \sigma_2) \quad dom(\psi) = ran(\iota_1) \quad dom(\psi) \cap ran(\iota_2) = \emptyset}{\sigma \vdash freeze_\psi(M) \downarrow (\langle \iota_2; o; (\rho + (o \circ \psi \circ \iota_1)) \rangle; \sigma_2)} \quad (5)$$

$$\textit{close} \quad \frac{\sigma \vdash M \downarrow (\langle \emptyset; o; x_i \overset{i \in I}{\mapsto} E_i \rangle; \sigma_2) \quad \forall i \in I,\ l_i \text{ fresh}}{\sigma \vdash close(M) \downarrow (\langle \emptyset; o; x_i \overset{i \in I}{\mapsto} l_i \rangle; \sigma_2[l_i \overset{i \in I}{\mapsto} E_i[l_j/x_j]_{j \in I}])} \quad (6)$$

$$\textit{projection} \quad \frac{\sigma \vdash M \downarrow (\langle \iota; o; \rho \rangle; \sigma_2) \quad \sigma_2 \vdash (\rho \circ o)(X) \downarrow (V; \sigma_3)}{\sigma \vdash M.X \downarrow (V; \sigma_3)} \quad (7)$$

$$\textit{location} \quad \frac{\sigma(l) = V}{\sigma \vdash l \downarrow (V; \sigma)} \quad (8)$$

$$\frac{\sigma(l) = E \quad \sigma[l \mapsto \texttt{error}] \vdash E \downarrow (V; \sigma_2)}{\sigma \vdash l \downarrow (V; \sigma_2[l \mapsto V])} \quad (9)$$

Figure 3. Semantics

expression repeatedly and to signal an error for cyclic definitions. On completion, the heap state is updated with the resulting value.

It may be useful to note that we do not need evaluation rules for names or identifiers, since in the normal evaluation names are substituted by expressions and identifiers by locations.

As one may have noticed, there is potential that evaluation gets stuck. One serious source is when projection is made from an open mixin. Instead of preventing such a scenario at the operational semantics level, we leave it to the type system to eliminate the possibilities for evaluation to get stuck. Although we do not present a type system in this paper, straightforward adaptation of previous work such as (Flatt and Felleisen 1998) is sufficient for this purpose. The type system would be able to eliminate other ill-typed scenarios such as an attempt to merge mixins with overlapping output names, i.e., $dom(o_1) \cap dom(o_2) \neq \emptyset$ in rule (2). We have omitted inference rules for propagating error states, assuming when evaluation encounters an `error` during deduction, it immediately terminates signaling a runtime error.

3.1 Call-by-name and eager variants

Small modifications to evaluation rules let the operational semantics model call-by-name and eager evaluation strategies. This subsection presents those variants.

To model a call-by-name strategy, we replace rules (8) and (9) with the single rule:

$$\frac{\sigma \vdash \sigma(l) \downarrow (V; \sigma_2)}{\sigma \vdash l \downarrow (V; \sigma_2)} \quad (10)$$

Heap states are not updated, thus expressions stored are re-evaluated whenever accessed.

By eager evaluation strategy, we mean an evaluation strategy that evaluates all components of a mixin at once when the mixin is closed. This is also easily implemented by replacing rule (6) with:

$$\frac{\begin{array}{c}\sigma \vdash M \downarrow (\langle \emptyset; o; x_i \overset{i\in I}{\mapsto} E_i \rangle; \sigma_2) \quad I = \{1, 2, \ldots, n\} \\ \forall i \in I,\ l_i \text{ fresh } \ \sigma' = \sigma_2[l_i \overset{i\in I}{\mapsto} E_i[l_j/x_j]_{j\in I}] \\ \sigma' \vdash l_1 \downarrow (V_1; \sigma_1') \quad \sigma_1' \vdash l_2 \downarrow (V_2; \sigma_2') \quad \cdots \quad \sigma_{n-1}' \vdash l_n \downarrow (V_n; \sigma_n')\end{array}}{\sigma \vdash close(M) \downarrow ([\emptyset; o; x_i \overset{i\in I}{\mapsto} l_i]; \sigma_n')} \quad (11)$$

4. Lazy mixins and disciplined effects

We extend the operational semantics of the previous section so that it takes account of constraints on the evaluation order of components of mixins. In examples of this section, we use a side-effecting function "print C", which prints the resulting value of evaluating C, then returns the value, to visualize the evaluation order.

4.1 Design space of evaluation strategies

There are several evaluation strategies that we found interesting to consider. Below we explain those strategies which we have in mind.

Firstly we consider the following example:

```
mixin M1 = close(
  { let c1 = print 1  let c2 = print 2
    let c3 = print 3  let c4 = print 4 })
```

Top-down strategy We may want components of a mixin to be evaluated following the textual definition order in which they appear in the source program. This constraint will ensure, in the above example, that 1 is printed before 2, and 2 is before 3, and 3 is before 4. This strategy is reminiscent of ML's strategy.

Lazy-field and lazy-record strategies If we adopt top-down strategy, we have a choice on whether to make accessible components of a mixin immediately after they are evaluated, while other components of the mixin are still to be evaluated consecutively. We call lazy-field strategy the strategy that allows such access and lazy-record strategy the strategy that does not allow. Intuitively lazy-field strategy treats a closed mixin as a record with lazy fields like { a1 = lazy (print 1); a2 = lazy (print 2) }, where "{" and "}" are record constructors of the core language here, while lazy-record strategy does as a lazy record like lazy ({ a1 = print 1; a2 = print 2 }). For instance, let's consider executing the program:

```
mixin M2 = close ({ let c1 = 1  let c2 = 2 * M3.c1 })
mixin M3 = close ({ let c1 = 3 + M2.c1 })
let main = M2.c2
```

We assume, for the explanatory purpose, that a program consists of a sequence of mixin definitions plus a core value definition named main, where the top-level mixin bindings can be mutually recursive. The execution of the program evaluates the defining expression of main. For the execution of the above program to succeed, M2.c1 should be accessible to M3.c1 before evaluation of M2.c2 is completed. Hence the execution succeeds with lazy-field strategy, but it fails with lazy-record strategy. Clearly lazy-field strategy is more permissive. Although restrictive, lazy-record strategy is interesting to consider particularly in the presence of finer grained accessibility control, as we will see below.

Internal and external accessibilities It can be useful to change accessibility to components of a mixin depending on whether the access is made inside the same mixin or outside. For instance, let us consider the following program:

```
mixin M4 = close(
  { let c1 = 1 + 2  let c2 = c1 + 4  let c3 = print "ok" })
let main = M4.c2
```

For the execution to succeed, M4.c1 should be accessible to M4.c2 immediately after M4.c1 is evaluated but before evaluation of M4.c2 is completed. Hence we need lazy-field strategy inside M4. However we may want components of M4 to become accessible outside, only after all components of M4 have been evaluated. In other words, we may want lazy-record strategy outside M4 to make sure "ok" is necessarily printed before any component of M4 becomes externally accessible. Indeed, in the widget mixins example of Section 2.2, we envisaged this internally-lazy-field externally-lazy-record strategy. For instance, the component menu of MyGUI should be accessible inside once it is created, to configure the widget containment hierarchy via functions setMenuItems and setMenus. However menu should not be accessed outside MyGUI, before the configuration is completed, i.e. all components, including anonymous ones, of MyGUI are evaluated. Internally-lazy-field externally-lazy-record strategy is also potentially interesting in the light of our experience in programming with ML modules; it is sometimes convenient to include anonymous side-effecting expressions at the end of a structure for the debugging purpose or to properly initialize mutable components of the structure.

As well as design choices among above strategies, we have a choice on how to propagate constraints when merging or closing mixins. In contrast to this vast design space, we do not have enough experience in programming with mixins. Plainly we cannot determine at present which strategy is most beneficial in practice. Hence we keep our formalization open to strategies. That is, the formalization is abstracted over constraints on strategies and can be instantiate to express particular strategies.

4.2 The constraint language

We use binary relations as our constraint language to express various evaluation strategies. We identify binary relations on any set P as sets of pairs of elements in P. For instance, the relation {(c1, c2) (c2, c3) (c3, c4)} expresses top-down strategy in the first example. The relation {(c1, c3) (c2, c4)} stipulates that c1 should be evaluated before c3 and c2 before c4. There is no constraint on the evaluation order between c1 and c2 or c3 and c2. Hence this

relation does not fix the evaluation order in a unique way. For instance in the first example, either of the output "1 2 3 4" or " 2 1 3 4" is compatible with the relation.

To deal with more expressive strategies, we distinguish three sorts of identifiers: (ordinary) identifiers x for controlling the evaluation order; *internal identifiers* $\hat{x}$ for internal accessibility; *external identifiers* $\overline{x}$ for external accessibility. For instance, the relation $\{(x_1, \hat{x_2})\}$ stipulates that the component x_1 must be evaluated before the component x_2 becomes accessible inside the same mixin. Similarly $\{(x_1, \overline{x_2})\}$ stipulates that x_1 must be evaluated before x_2 becomes accessible outside. We will formalize how relations on these three sorts of identifiers control the evaluation order and accessibility later, when we define the operational semantics. Below we give a descriptive explanation on how those three sorts of identifiers can be used to express strategies we proposed above.

If we assign the relation $\{(\mathsf{c1}, \mathsf{c2})\}$ as the constraint to M2 in the second example, the execution of the program succeeds. The relation stipulates nothing about how M2.c1 becomes accessible, implying M2.c1 is accessible both inside M2 and outside immediately after it is evaluated. If we assign the relation $\{(\mathsf{c1}, \mathsf{c2}), (\mathsf{c2}, \overline{\mathsf{c1}})\}$ as the constraint to M2, the execution fails. The relation stipulates that M2.c1 becomes accessible outside M2 only after M2.c2 has been evaluated. However M3.c1 needs to access M2.c1 to be evaluated, and M2.c2 to M3.c1; there is a circular dependency.

Top-down internally-lazy-field externally-lazy-record strategy in the third example is expressed by assigning the relation $\{(\mathsf{c1}, \mathsf{c2}), (\mathsf{c2}, \mathsf{c3}), (\mathsf{c3}, \overline{\mathsf{c1}}), (\mathsf{c3}, \overline{\mathsf{c2}})\}$ as the constraint to M4. We remark that there are implicit constraints imposed by the semantics, such as $(\mathsf{c1}, \hat{\mathsf{c1}})$ and $(\mathsf{c1}, \overline{\mathsf{c1}})$ (but not $(\hat{\mathsf{c1}}, \overline{\mathsf{c1}})$). The reason is simple: the component c1 must be evaluated before it becomes accessible. Hence if the relation included $(\overline{\mathsf{c1}}, \mathsf{c1})$, which introduces a circular dependency together with the implicit constraint $(\mathsf{c1}, \overline{\mathsf{c1}})$, the evaluation would fail. Precisely, the operational semantics signals an error. The relation $\{(\mathsf{c1}, \mathsf{c2}), (\mathsf{c2}, \mathsf{c3}), (\mathsf{c3}, \hat{\mathsf{c1}}), (\mathsf{c3}, \hat{\mathsf{c2}}) (\mathsf{c3}, \overline{\mathsf{c1}}), (\mathsf{c3}, \overline{\mathsf{c2}})\}$ in the third example expresses top-down internally-lazy-record externally-lazy-record strategy. We have omitted including constraints such as $(\mathsf{c2}, \hat{\mathsf{c1}})$, since it is anyway induced from $(\mathsf{c2}, \mathsf{c3})$ and $(\mathsf{c3}, \hat{\mathsf{c1}})$ by transitivity. Transitivity is imposed by the semantics: if c2 must be evaluated before c3, and c3 before $\hat{\mathsf{c1}}$, then naturally c2 must be evaluated before $\hat{\mathsf{c1}}$.

4.3 Operational semantics

We extend a structure with a binary relation on ordinary, internal and external identifiers. Precisely, a structure $[\iota; o; \rho; \theta]$ is now 4-tuple of an input assignment ι, output assignment o, local binding ρ, and *local constraint* θ, where θ is a binary relation on $dom(\iota) \cup dom(\rho) \cup \{\overline{x} \mid x \in dom(\iota) \cup dom(\rho)\} \cup \{\hat{x} \mid x \in dom(\iota) \cup dom(\rho)\}$. Otherwise the syntax is unchanged from Figure 2. We use $\mathcal{X}$ as a metavariable for sets of ordinary, internal and external identifiers.

We assume given two functions $\mu(\mathcal{X}_1, \theta_1)$ and $\nu(\mathcal{X}_1, \theta_1, \mathcal{X}_2, \theta_2)$, where θ_1 and θ_2 are binary relations on $\mathcal{X}_1$ and $\mathcal{X}_2$ respectively. $\mu(\mathcal{X}_1, \theta_1)$ returns a binary relation on $\mathcal{X}_1$ and $\nu(\mathcal{X}_1, \theta_1, \mathcal{X}_2, \theta_2)$ does on $\mathcal{X}_1 \cup \mathcal{X}_2$. The operational semantics uses $\mu(\mathcal{X}_1, \theta_1)$ to build local constraints for mixins instantiated by the close operation, where $\mathcal{X}_1$ is the set of deferred and defined identifiers and θ_1 the local constraint of the operand mixin. Similarly it uses $\nu(\mathcal{X}_1, \theta_1, \mathcal{X}_2, \theta_2)$ to build local constraints for mixins composed by the sum operation, where $\mathcal{X}_1$ (resp. $\mathcal{X}_2$) is the set of deferred and defined identifiers and θ_1 (resp. θ_2) is the local constraints of the left (resp. right) right operand mixin. By not fixing the interpretations of μ or ν, we keep the formalization neutral of how to propagate local constraints when closing or merging mixins.

In Figure 4, we present the operational semantics which takes account of constraints. We use Θ as a metavariable for *global constraints*, or binary relations on locations. The judgment $\sigma; \Theta \vdash E \downarrow (V; \sigma'; \Theta')$ means that under global constraint Θ with heap state σ expression E evaluates into V, where the heap state and global constraint have evolved into σ' and Θ'. As well as heap states, global constraints evolve during evaluation, since new constraints are added when a mixin is closed (rule (17)). The notation $\Theta \backslash \{(l', l)\}$ denotes set subtraction. The notation $l \not\prec \Theta$ denotes the condition that there is not l' such that $(l', l) \in \Theta$.

Compared to previous evaluation rules in Figure 3, modifications are made on rules (2) (6), and (9), which we explain in turn.

As explained above, the sum operation uses the function ν to build a local constraint for the composed mixin (rule (13)).

When closing a mixin (rule (17)), three fresh locations l_i, l'_i, l''_i are created for each defined component x_i of the mixin, in order to separately control the evaluation order by l_i's, internal accessibility by l'_i's, and external accessibility by l''_i's. Four important points to be understood are as follows: 1) The local binding of the resulting mixin maps x_i's to l''_i's, i.e. locations for external access; 2) Evaluation of E_i is suspended at l_i, where x_i's in E_i are substituted by l'_i's, i.e. locations for internal access; 3) The new heap state maps l'_i's and l''_i's for internal and external accesses to l_i's to connect the accesses to underlying expressions; 4) The local constraint θ is transported to the global constraint, by instantiating ordinary, internal and external identifiers to the corresponding locations. The notation $\theta[l_i/x_i]_{i\in I}[l'_i/\hat{x_i}]_{i\in I}[l''_i/\overline{x_i}]_{i\in I}$ denotes the binary relation on locations obtained from θ by substituting l_i's for x_i's , l'_i's for $\hat{x_i}$'s, and l''_i's for $\overline{x_i}$'s, where the substitution is performed by regarding a binary relation as a set of pairs. The local constraint for the resulting mixin is build by the function μ.

We have introduced two new rules (20) and (21) for evaluating locations, replacing the previous rule (9), to take the global constraint into account. Rule (20) considers the case where the global constraint stipulates that l' should be evaluated before l, as described by the side-condition $(l', l) \in \Theta$. The rule evaluates l' first, while updating the heap state to map l to `error` and removing (l', l) from Θ; if the evaluation of E' attempts to evaluate l against the global constraint, an error is signaled. On completion of the evaluation, E is restored at l and the rule retries to evaluate l. The same rule (20) may be applied again, when there is another location which should be evaluated before l. Otherwise rule (21) is applicable. When there is no location which must be evaluated before l, as described by the side condition $l \not\prec \Theta$ in rule (21), then E is evaluated immediately.

It shall be informative to note that in rule (20) the heap state σ is updated with `error` to enforce the evaluation order stipulated by the constraint, while in rule (21) to detect ill-founded recursion. An important ingredient of the formalization is that evaluation of projection $M.X$ always goes through a location. This facilitates to control the evaluation order of components of mixins, in terms of a binary relation on locations. It is easily proved that if $\sigma(l) = V$ then $l \not\prec \Theta$. We also remark that the operational semantics, in particular the evaluation order, is not necessarily deterministic, depending on the constraint. For instance, if we assign a local constraint $\{(\mathsf{c1}, \mathsf{c3}), (\mathsf{c2}, \mathsf{c4})\}$ to the mixin M1 of the first example in Section 4.1, we can build deductions which result in the outputs "1 2 3 4" and "2 1 3 4". Non-determinism is not abnormal, but should be thought of as underspecification like the underspecified evaluation order of function parameters in some programming languages.

The operational semantics can be proved sane in the sense that evaluation does not diverge due to ill-founded recursion. In other words, when evaluation diverges, heap states are extended infinitely. A heap state is extended only when a mixin is closed (rule (17)). Hence the heap explosion implies that the close operation is

$$\textit{mixin structure}\quad \frac{}{\sigma;\Theta \vdash [\iota;o;\rho;\theta] \downarrow (\langle \iota;o;\rho;\theta\rangle;\sigma;\Theta)} \quad (12)$$

$$\textit{sum}\quad \frac{\sigma;\Theta \vdash M_1 \downarrow (\langle \iota_1;o_1;\rho_1;\theta_1\rangle;\sigma_2;\Theta_2) \quad \sigma_2;\Theta_2 \vdash M_2 \downarrow (\langle \iota_2;o_2;\rho_2;\theta_2\rangle;\sigma_3;\Theta_3) \quad dom(\rho_1)\cap dom(\rho_2)=\emptyset \quad dom(\iota_1)\cap dom(\iota_2)=\emptyset}{\sigma;\Theta \vdash M_1 \leftarrow M_2 \downarrow (\langle \iota_1+\iota_2;o_1+o_2;\rho_1+\rho_2;\nu(dom(\iota_1)\cup dom(\rho_1),\theta_1,dom(\iota_2)\cup dom(\rho_2),\theta_2)\rangle;\sigma_3;\Theta_3)} \quad (13)$$

$$\textit{rename}\quad \frac{\sigma;\Theta \vdash M \downarrow (\langle \iota;o;\rho;\theta\rangle;\sigma_2;\Theta_2)}{\sigma;\Theta \vdash \phi_1 \lhd M \rhd \phi_2 \downarrow (\langle \phi_1\circ\iota;o\circ\phi_2;\rho;\theta\rangle;\sigma_2;\Theta_2)} \quad (14)$$

$$\textit{hide}\quad \frac{\sigma;\Theta \vdash M \downarrow (\langle \iota;o;\rho\rangle;\sigma_2;\Theta_2)}{\sigma;\Theta \vdash \mathit{hide}_X(M) \downarrow (\langle \iota;o\backslash X;\rho\rangle;\sigma_2;\Theta_2)} \quad (15)$$

$$\textit{freeze}\quad \frac{\sigma;\Theta \vdash M \downarrow (\langle \iota_1+\iota_2;o;\rho;\theta\rangle;\sigma_2;\Theta_2) \quad dom(\psi)=ran(\iota_1) \quad dom(\psi)\cap ran(\iota_2)=\emptyset}{\sigma;\Theta \vdash \mathit{freeze}_\psi(M) \downarrow (\langle \iota_2;o;\rho+(o\circ\psi\circ\iota_1);\theta\rangle;\sigma_2;\Theta_2)} \quad (16)$$

$$\textit{close}\quad \frac{\begin{array}{c}\sigma;\Theta \vdash M \downarrow (\langle \emptyset;o;x_i \overset{i\in I}{\mapsto} E_i;\theta\rangle;\sigma_2;\Theta_2) \\ \forall i\in I,\ l_i\ l_i'\ l_i'' \text{ fresh} \quad \Theta_3=\theta[l_i/x_i]_{i\in I}[l_i'/\hat{x}_i]_{i\in I}[l_i''/\overline{x_i}]_{i\in I}\end{array}}{\sigma;\Theta \vdash \mathit{close}(M) \downarrow (\langle \emptyset;o;x_i \overset{i\in I}{\mapsto} l_i'';\mu(\{x_i \mid i\in I\},\theta)\rangle;\sigma_2[l_i \overset{i\in I}{\mapsto} E_i[l_j'/x_j]_{j\in I}][l_i' \overset{i\in I}{\mapsto} l_i][l_i'' \overset{i\in I}{\mapsto} l_i];\Theta_2\cup\Theta_3)} \quad (17)$$

$$\textit{projection}\quad \frac{\sigma;\Theta \vdash M \downarrow (\langle \iota;o;\rho;\theta\rangle;\sigma_2;\Theta_2) \quad \sigma_2;\Theta_2 \vdash (\rho\circ o)(X) \downarrow (V;\sigma_3;\Theta_3)}{\sigma;\Theta \vdash M.X \downarrow (V;\sigma_3;\Theta_3)} \quad (18)$$

$$\textit{location}\quad \frac{\sigma(l)=V}{\sigma;\Theta \vdash l \downarrow (V;\sigma;\Theta)} \quad (19)$$

$$\frac{(l',l)\in\Theta \quad \sigma(l)=E \quad \sigma[l\mapsto \texttt{error}];\Theta\backslash\{(l',l)\} \vdash l' \downarrow (V';\sigma_2;\Theta_2) \quad \sigma_2[l\mapsto E];\Theta_2 \vdash l \downarrow (V;\sigma_3;\Theta_3)}{\sigma;\Theta \vdash l \downarrow (V;\sigma_3;\Theta_3)} \quad (20)$$

$$\frac{l \not\in \Theta \quad \sigma(l)=E \quad \sigma[l\mapsto \texttt{error}];\Theta \vdash E \downarrow (V;\sigma';\Theta')}{\sigma;\Theta \vdash l \downarrow (V;\sigma'[l\mapsto V];\Theta')} \quad (21)$$

Figure 4. Semantics with constraint

used infinitely often. This is similar to a situation where infinite recursion of function calls exhausts the stack. The proof of this property is easy and found in (Nakata 2008a).

4.4 An example

We present an example by instantiating the constraint to express one particular evaluation strategy. The strategy is motivated by our previous work on examining a lazy evaluation strategy for recursive ML-style module (Nakata 2008b). The purpose of the example is not to advocate this particular strategy, but to deliver the better intuition about how to use the constraint.

For a structure $[\iota;o;\rho;\theta]$, we let θ be the relation:

$$\{(x_i,x_j) \mid x_i \in dom(\rho)\cap IdsC, x_j \in dom(\rho), i<j\} \cup \{(x_i,\overline{x_j}) \mid x_i \in (dom(\iota)\cup dom(\rho))\cap IdsC, x_j \in dom(\iota)\cup dom(\rho)\}$$

Above we have assumed for any x_i, x_j, if $i<j$ then the definition or declaration for x_i textually precedes that for x_j in the source program. *IdsC* denotes the set of identifiers bounds to core expressions. I.e. we assume *Idents* consists of disjoint sets of identifiers to be bound to core expressions and mixin expressions, respectively.

We give interpretations to functions μ and ν as follows:

$$\mu(\mathcal{X},\theta)=\emptyset$$
$$\nu(\mathcal{X},\theta,\mathcal{X}',\theta') = \theta \cup \theta' \cup \{(x_i,\overline{x_j} \mid x_i \in (\mathcal{X}\cup\mathcal{X}')\cap IdsC, x_j \in \mathcal{X}\cup\mathcal{X}'\}$$

The strategy is close to top-down internally-lazy-field externally-lazy-record strategy, where sub-mixins are evaluated lazily. Below we explain the strategy in detail.

1. Preceding core components of the same and enclosing mixins are evaluated first. Enclosing mixins may contain other sub-mixins, whose core fields need not be evaluated first. This is enforced by including in the local constraint of a structure the relation $\{(x_i,x_j) \mid x_i \in dom(\rho)\cap IdsC, x_j \in dom(\rho), i<j\}$. Note that x_j may be bound to a mixin expression, and in order to evaluate the expression all core components preceding to it must have been evaluated. Thus the constraint controls the evaluation order not only of core components of the same mixin but also of those of enclosing mixins.
2. Components of a mixin become accessible inside the same mixin immediately after they are evaluated, but are accessible outside only after all the core components have been evaluated. This is enforced by not having constraints mentioning internal identifiers and by including in the local constraint of a structure the relation $\{(x_i,\overline{x_j}) \mid x_i \in (dom(\iota)\cup dom(\rho))\cap IdsC, x_j \in dom(\iota)\cup dom(\rho)\}$. Observe how we propagate the constraint on external accessibility when merging mixins by including the relation $\{(x_i,\overline{x_j} \mid x_i \in (\mathcal{X}\cup\mathcal{X}')\cap IdsC, x_j \in \mathcal{X}\cup\mathcal{X}'\}$ in the result of ν.

3. While core components are evaluated following the textual definition order, there is no constraint on the evaluation order between components of merged mixins; ν does not introduce (x_i, x_j) or (x_i, x_j) for any $x_i \in \mathcal{X}$ and $x_j \in \mathcal{X}'$.
4. We do not leave any constraint after a mixin is closed; it is enough to enforce the evaluate order once. This is reflected in the interpretation of μ.

As we have said, we do not intend to justify ourselves in choosing this strategy, because we do not have enough programming experience to do so. Nevertheless we motivate the strategy below, aiming at posing questions to readers about possible concerns one may face in the quest of better design choices.

The first condition ensures that backward references to core components of the same and enclosing mixins are necessarily proper values, but not suspensions or `error`'s. This provides programmers with a safety guarantee on backward references to core components, which seems useful for ML-initiated programmers. Interestingly this design choice also bears a similarity to Java's class initialization policy (Gosling et al. 2005). We have already motivated in Section 4.1 the combination of internally-lazy-field and externally-lazy-record strategies; internally-lazy-field strategy leaves flexibility in intra-mixin recursion, while externally-lazy-record strategy keeps the evaluation stable towards outside. A possible drawback of externally-lazy-record strategy, compared to externally-lazy-field strategy, is that we may loose flexibility in inter-mixin recursion. However we are less concerned by inter-mixin recursion, since we believe mixins are designated to support flexible intra-mixin recursion; notably the sum and freeze operations are useful for taking fix-points inside a mixin.

We are less confident in the choice of ν. But it could be useful to keep the evaluation order independent between separately defined mixins; it is unlikely that a programmer can foresee in which order components of other mixins to be merged with should/could be evaluated.

4.5 An extension

There is a strategy which we want to consider, but our constraint language is not expressive enough for it. The strategy is a variant on top-down internally-lazy-field externally-lazy-field strategy, where we impose an extra constraint, called *trigger-constraint*, that all components of a mixin must be evaluated at once before the first access to a component of the mixin returns. For instance, let us consider the following program:

```
mixin M1 = {
  let c1 = print 1  let c2 = M2.c2
  let c3 = print (c1 + c2)  let c4 = print 5 }
mixin M2 = {
  let c1 = M1.c1  let c2 = print (c1 + 1)  let c3 = print 4 }
let main = M1.c3
```

According to the above proposed strategy, the execution succeeds and "1 2 4 3 5" is printed. (Recall that "print (1+2)" returns 3, after printing 3). Here is why.

1. Top-down strategy ensures the printing orders "1 3 5" and "2 4" independently.
2. Thanks to internally-lazy-field strategy, M1.c3 and M2.c2 are successfully evaluated.
3. Thanks to externally-lazy-field strategy, evaluation of M2.c1 succeeds. Note that M2 is forced to evaluate by the access from M1.c2. Hence with externally-lazy-record strategy, the evaluation fails, since M1.c1 is not yet accessible outside then.
4. Finally and importantly, the trigger-constraint ensures that M2.c3 is evaluated before evaluation of M1.c2 returns and that M1.c4 is before evaluation of main returns. This explains why 4 is printed immediately after 2, and 5 is after 3.

This strategy comes from our previous work on lazy recursive modules (Nakata 2008b). In a recursive modules setting, inter-module recursion is important, hence we wanted externally-lazy-field strategy. We found the trigger-constraint useful to enforce our design policy that once a module is accessed, all its component are eventually evaluated. We want to consider the same strategy as our previous proposal in a mixin context, too. Observe that the trigger-constraint only lets the access to M1.c3 trigger evaluation of M1.c4, but M1.c1 is already accessible both inside and outside once it is evaluated. We cannot include any of constraints (c4, c3), (c4, $\overline{\text{c3}}$), or (c4, $\hat{\text{c3}}$) in the local constraint of M1, since the first one is inconsistent with top-down strategy, the second with externally-lazy-field strategy, the third with internally-lazy-field strategy.

4.5.1 Formalization

To gain extra expressivity to deal with the trigger-constraint, we extend a local constraint to be a pair of a binary relation on identifiers and a set of sets of identifiers. Precisely, a structure $[\iota; o; \rho; \pi]$ now contains a local constraint π, which is a pair (θ, δ) of a binary relation θ on $dom(\iota) \cup dom(\rho) \cup \{\overline{x} \mid x \in dom(\iota) \cup dom(\rho)\} \cup \{\hat{x} \mid x \in dom(\iota) \cup dom(\rho)\}$ and a set δ of sets of elements in $dom(\iota) \cup dom(\rho)$. θ expresses constraints on the evaluation order and accessibilities as before. δ expresses the trigger-constraint; if $\mathcal{X}$ is in δ, then evaluation of all components bound to identifiers in $\mathcal{X}$ is triggered at once when any of the components is accessed for the first time. For instance when δ contains $dom(\rho)$, then all defined components of the structure, but deferred ones, are evaluated at once when any of the defined components is accessed for the first time. Accordingly, a global constraint Π is now a pair (Θ, Δ) of a binary relation Θ on locations and set Δ of sets of locations. The functions μ and ν need to be extended to operate on pairs of a binary relation and set of sets.

To take account of the trigger-constraint, We replace rules (20) and (21) by rules (22), (23) and (24), and rule (17) by rule (25) as given in Figure 5. The notation $l \notin \Delta$ denotes the condition that Δ does not contain a set containing l.

The additional work is to check, before evaluating a location l, whether there are locations whose evaluation is triggered by l. Rule (22) considers the case where l triggers evaluation of l_i's, as described by the side condition $\{l, l_1 \dots l_n\} \in \Delta$. The rule bears responsibility for evaluating l_i's and $\{l, l_1 \dots l_n\}$ is discharged from Δ. Except for the side condition $l \notin \Delta$, rules (23) and (24) are identical to previous rules (20) and (21); the side condition ensures that all trigger constraints involving l have been handled. It is easily proved that (19) and (22) are exclusive to each other. Rule (25) is a technical adjustment, since the rule now needs to transport a pair of a binary relation and set of sets from the local constraint to the global constraint. The notation $\delta[l_i/x_i]_{i\in I}[l'_i/\hat{x}_i]_{i\in I}[l''_i/\overline{x_i}]_{i\in I}$ denotes the set of sets of locations obtained from δ by substituting l_i's for x_i's , l'_i's for $\hat{x_i}$'s, and l''_i's for $\overline{x_i}$'s. We do not repeat the other rules; the necessary modification is to replace Θ's by Π's.

4.5.2 An example

As an example of how to use the trigger-constraint, we present a possible object initialization strategy in class-based object-oriented languages. The purpose of the example is not to explain an encoding of object-oriented language features such as inheritance and overriding, which is already examined in previous work (Ancona and Zucca 2002). Here we focus on the aspect of object initialization, by regarding components of a mixin as (instance) fields of an object. In this scenario, open mixins correspond to classes and closed mixins to objects. We shall not need the full expressivity of *Lyre* to model standard class-based object-oriented languages. For

$$\frac{\Pi = (\Theta, \Delta) \quad \{l, l_1 \ldots l_n\} \in \Delta \quad \sigma; (\Theta, \Delta \backslash \{\{l, l_1 \ldots l_n\}\}) \vdash l \downarrow (V; \sigma'; \Pi') \quad \sigma'; \Pi' \vdash l_1 \downarrow (V_1; \sigma'_1; \Pi'_1) \quad \sigma'_1; \Pi'_1 \vdash l_2 \downarrow (V_2; \sigma'_2; \Pi'_2) \quad \cdots \quad \sigma'_{n-1}; \Pi'_{n-1} \vdash l_n \downarrow (V_n; \sigma'_n; \Pi'_n)}{\sigma; \Pi \vdash l \downarrow (V; \sigma'_n; \Pi'_n)} \quad (22)$$

$$\frac{l \notin \Delta \quad (l', l) \in \Theta \quad \sigma(l) = E \quad \sigma[l \mapsto \texttt{error}]; (\Theta \backslash \{(l', l)\}, \Delta) \vdash l' \downarrow (V'; \sigma_2; \Pi_2) \quad \sigma_2[l \mapsto E]; \Pi_2 \vdash l \downarrow (V; \sigma_3; \Pi_3)}{\sigma; (\Theta, \Delta) \vdash l \downarrow (V; \sigma_3; \Pi_2)} \quad (23)$$

$$\frac{\Pi = (\Theta, \Delta) \quad l \notin \Delta \quad l \not\prec \Theta \quad \sigma(l) = E \quad \sigma[l \mapsto \texttt{error}]; \Pi \vdash E \downarrow (V; \sigma'; \Pi')}{\sigma; \Pi \vdash l \downarrow (V; \sigma'[l \mapsto V]; \Pi')} \quad (24)$$

$$\frac{\begin{array}{c}\sigma; \Pi \vdash M \downarrow (\langle \emptyset; o; x_i \overset{i \in I}{\mapsto} E_i; \pi \rangle; \sigma_2; \Pi_2) \quad \pi = (\theta, \delta) \quad \Pi_2 = (\Theta, \Delta) \\ \forall i \in I,\ l_i\ l'_i\ l''_i \text{ fresh} \quad \Theta' = \theta[l_i/x_i]_{i \in I}[l'_i/\hat{x_i}]_{i \in I}[l''_i/\overline{x_i}]_{i \in I} \quad \Delta' = \delta[l_i/x_i]_{i \in I}[l'_i/\hat{x_i}]_{i \in I}[l''_i/\overline{x_i}]_{i \in I}\end{array}}{\sigma; \Pi \vdash close(M) \downarrow (\langle \emptyset; o; x_i \overset{i \in I}{\mapsto} l''_i; \mu(\{x_i \mid i \in I\}, \pi) \rangle; \sigma_2[l_i \overset{i \in I}{\mapsto} E_i[l'_j/x_j]_{j \in I}][l'_i \overset{i \in I}{\mapsto} l_i][l''_i \overset{i \in I}{\mapsto} l_i]; (\Theta \cup \Theta', \Delta \cup \Delta'))} \quad (25)$$

Figure 5. Extension with trigger-constraint

```
class A { val a1 = ... }
class B extends A { val b1 = ... val b2 = ... }
class C extends B { val c1 = .. }
let c = new C
```

Figure 6. A class hierarchy

specificity, we restrict ourselves to a fragment of the calculus satisfying the following conditions:

- Structures do not contain sub-mixins. This implies we do not consider inner classes.
- The sum operation only takes open mixins as operands. The operation corresponds to inheritance. Then this is standard, since usually classes do not inherit from objects, or vice versa. In the sum construct $M_1 \leftarrow M_2$, we assume the left operand mixin M_1 corresponds to the superclass and the right operand mixin M_2 to the inheriting class.

For a structure $[\iota; o; \rho; (\theta, \delta)]$, we assign internally-lazy-record externally-lazy-record strategy. The initialization order of fields within an object is kept unspecified. That is, we let θ be the relation:

$$\{(x_i, \overline{x_j}) \mid x_i, x_j \in (dom(\iota) \cup dom(\rho))\} \\ \cup \{(x_i, \hat{x_j}) \mid x_i, x_j \in (dom(\iota) \cup dom(\rho))\}$$

We let δ be the singleton:

$$\{(dom(\iota) \cup dom(\rho))\}$$

The trigger-constraint δ above stipulates that all the fields of an object must be initialized at once.

We give interpretations to functions μ and ν as follows:

$$\mu(\mathcal{X}, \pi) = (\emptyset, \emptyset) \\ \nu(\mathcal{X}, (\theta, \delta), \mathcal{X}', (\theta', \delta')) = \\ \quad (\theta \cup\ \theta' \cup\ \{(x_i, x_j) \mid x_i \in \mathcal{X}, x_j \in \mathcal{X}'\},\ \{(\mathcal{X} \cup \mathcal{X}')\})$$

The interpretation of ν makes the strategy interesting. Viewed as object initialization, the strategy is described as follows.

1. Fields are initialized following the class hierarchy. That is, fields of superclasses are initialized before those of sub-classes. For instance in Figure 6, the field a1 of the object c is initialized before b1 or b2, and b1 and b2 are before c1. This is imposed by including the constraint $\{(x_i, x_j) \mid x_i \in \mathcal{X}, x_j \in \mathcal{X}'\}$ in the result of the first element of ν.
2. Any field inherited from a superclass becomes accessible both inside and outside, once all the fields from the superclass are initialized. Hence in the example, c.b1 and c.b2 become accessible immediately after both have been initialized, but before c.c1 is initialized. Note that neither of c.b1 nor c.b2 is accessible even inside before both have been initialized. This is specified by the local constraint assigned to a structure.
3. We assume the new operation for instantiating objects from classes is equivalent to the close operation followed by access to some field of the mixin closed. For instance we may assume the *Object* class which resides in the root of the class hierarchy, i.e. a superclass of any class, contains a special field named init for that purpose. Then the strategy enforces the standard object initialization policy where all fields of an object are initialized when it is created. This is where we use the trigger-constraint. The result $\{(\mathcal{X} \cup \mathcal{X}')\}$ of the second element of ν ensures that all field of an object are initialized at once when init is accessed, since it triggers initialization of the fields of inheriting classes, when the init field is initialized.

In fact this strategy, in particular the characteristic described second above, is inspired by the object initialization strategy of the F# programming language (Syme and Margetson) [5]. F# places restrictions on possible object initialization patterns, enhancing safety by eliminating programming styles which are often awkward sources for null-pointer exceptions. As a result, the strategy is less flexible than that of some object-oriented languages such as Java, yet it can still support common programming idioms found in object-oriented programming. In short, we found F#'s strategy an interesting example of disciplined evaluation orders.

5. Related Work

One difference between previous work and the present work is that each of previous work examined an evaluation strategy, while we explored the design space of lazy evaluation strategies and give the operational semantics which can deal with several strategies. Most related to our work is Ancona and Zucca's call-by-name mixin calculus (Ancona and Zucca 2002) and Hirschowitz and Leroy's call-by-value mixin calculus (Hirschowitz and Leroy 2005). Ancona et al. investigated the interaction between mixins and computational effect, using a monadic metalanguage as semantic basis (Ancona et al. 2003). We will review the three in detail below. S. Fagorzi and E. Zucca proposed R-calculus to allow projection from open mixins in a consistent way (Fagorzi and Zucca 2007); this is a design direction we have not explored. Our previous work proposed a lazy evaluation strategy for recursive ML-style modules, which we explained in Section 4.5. Design questions we encountered there motivated strategies we proposed in Section 4.1. We do not look back at the history of mixins, but only mention a few of influential papers (Bracha 1992; Ancona and Zucca 2002; Hirschowitz and

[5] We believe the strategy presented is close to F#'s. Yet the exact strategy is under-specified in the language documentation.

Leroy 2005; Duggan and Sourelis 1996; Flatt and Felleisen 1998). In particular, type systems which eliminate unsound scenarios such as to close mixins having holes or to select a component from an open mixin, have been well-investigated. Those type system proposals are orthogonal and complementary to the presented work.

Call-by-name mixins Ancona and Zucca formalized a mixin calculus, named *CMS*, with call-by-name evaluation using small-step semantics (Ancona and Zucca 2002). The formalization is concise and allows equational reasoning of mixins. However the call-by-name semantics might not be suitable to be used with a call-by-value core language allowing arbitrary side-effects, such as the ML core language, since with the call-by-name semantics evaluation of let rec x = x diverges when x is selected and the same side-effect can be produced repeatedly. Large part of the formalization in Section 3 is borrowed from *CMS*. We adapted their small-step semantics to big-step semantics. Technically we made a small but important modification; in *CMS* an output assignment maps names to expressions, whereas in *Lyre* names to identifiers. In this way we avoid duplicating expressions in the freeze operation.

Call-by-value mixins Hirschowitz and Leroy formalized a mixin calculus with call-by-value evaluation (Hirschowitz and Leroy 2005). Generally call-by-value mixins result in the simplest evaluation order, while lazy mixins are more flexible in handling recursion. For instance, the marshallers example in Section 2.3 relies on the laziness to specify marshallers for recursive data types. We are motivated to distinguish open and closed mixins by their work. Apart from the difference of call-by-value and lazy, *Lyre* differs from their calculus in the sum operation in that we allow mixins to be merged independently of whether they are closed or open, while they only consider the sum operation on open mixins. Our design choice is motivated to keep flexibility in sharing side effects. For instance, it was useful in MakeSet and MakeMultiSet mixins example from Section 2.1; the sharing of counter between Set and MultiSet is achieved by merging the closed mixin Key with MakeSet and MakeMultiSet. It should be noted that one important objective of Herschowitz and Leroy's work is to statically ensure initialization safety of mixins. The objective of our paper is not about static guarantees of initialization safety.

Effectful mixins and equational reasoning Ancona et al. examined the interaction between mixins and computational effect, by means of a recursive monadic binding (Ancona et al. 2003). The sum operation only takes open mixins as argument in their calculus. They separate computational components from non-computational ones, where the former are evaluated exactly once, while the latter are re-evaluated whenever they are projected. Computational components of a mixin are evaluated at once when the mixin is closed by the *doall* operation, in a similar way to the rule (11) from Section 3.1. To the best of our understanding, their evaluation strategy is top-down internally-lazy-field externally-lazy-record strategy, where closed sub-mixins are evaluated immediately. When mixins are merged, the computational components of the right-operand mixin are evaluated before those of the left-operand mixin. The separation of computational components lets them retain the *CMS* equational reasoning.

6. Conclusion

We have formalized the operational semantics for a mixin calculus with lazy evaluation. We started by considering a simpler semantics where a component of a mixin is evaluated when it is projected. Then we extended the semantics to impose various restrictions on the evaluation order of and accessibilities to components of mixins, by adding local constraint to mixin structures and global constraint to the evaluation judgment. We have kept the formalization neutral of constraints and it can be instantiated to express several evaluation strategies. As well as we considered the design space of strategies, we took a closer look at two particular strategies.

We do not claim the formalization is expressive enough to deal with all interesting evaluation strategies. At the same time, we do not think it technically difficult to extend the formalization to express more strategies. As the extension we made in Section 4.5 exemplifies, we may well gain more expressivity by extending the constraint language and by adding more evaluation rules. Indeed a moot point was to keep the formalization not too complicated without giving up too much expressivity to deal with interesting strategies.

Acknowledgments

I am grateful to Xavier Leroy for the valuable advice and fruitful discussions throughout the development of this work.

References

D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 12, 2002.

D. Ancona, S. Fagorzi, E. Moggi, and E. Zucca. Mixin Modules and Computational Effects. In *Proc. ICALP*, 2003.

G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple inheritance*. PhD thesis, University of Utah, 1992.

L. Cardelli. Program Fragments, Linking, and Modularization. In *Proc. POPL*, pages 266–277, 1997.

K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. PLDI*, pages 50–63, 1999.

D. Duggan and C. Sourelis. Mixin modules. In *Proc. ICFP*, 1996.

S. Fagorzi and E. Zucca. A Calculus of Open Modules: Call-by-need Strategy and Confluence,. *Mathematical Structures in Computer Science*, 17, 2007.

K. Fisher and J. Reppy. The design of a class mechanism for moby. In *Proc. PLDI*. ACM Press, 1999.

M. Flatt and M. Felleisen. Units: Cool Moduels for HOT Languages. In *Proc. PLDI*. ACM Press, 1998.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*, chapter 12.4. Prentice Hall, 2005.

T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. *ACM Transactions on Programming Languages and Systems*, 27(5): 857–881, 2005.

A. Kennedy. Functional Pearls: Pickler Combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

M. Lillibridge and R. Harper. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proc. POPL*, pages 123–137, 1994.

K. Nakata. Lazy mixins and disciplined effects. Longer version available at `http://pauillac.inria.fr/~nakata/Lyre.pdf`, 2008a.

K. Nakata. Frozen Modules:A lazy evaluation strategy for more recursive initialization patterns. Draft paper available at `http://pauillac.inria.fr/~nakata/`, 2008b.

M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP*, 2003.

S. Owens and M. Flatt. From Structures and Functors to Modules and Units. In *Proc. ICFP*, 2006.

C. Russo. Recursive Structures for Standard ML. In *Proc. ICFP*, pages 50–61. ACM Press, 2001.

D. Syme. Initializing mutually referential abstract objects: The value recursion challenge. In *Proc. Workshop on ML*, volume 148, pages 3–25, 2005.

D. Syme and J. Margetson. The F# Programming Language. Software and documentation available on the Web, `http://research.microsoft.com/fsharp/fsharp.aspx`.

M. Torgersen. The Expression Problem Revisited. In *Proc. ECOOP*, volume 3086, 2004.

$$\frac{\{l, l_1 \dots l_n\} \in \Delta \quad \sigma; (\Theta, \Theta', \Delta \backslash \{\{l, l_1 \dots l_n\}\}, \Delta' \cup \{\{l, l_1 \dots l_n\}\}) \vdash l \downarrow (V; \sigma'; \Pi') \quad \sigma'; \Pi' \vdash l_1 \downarrow (V_1; \sigma'_1; \Pi'_1) \quad \sigma'_1; \Pi'_1 \vdash l_2 \downarrow (V_2; \sigma'_2; \Pi'_2) \cdots \sigma'_{n-1}; \Pi'_{n-1} \vdash l_n \downarrow (V_n; \sigma'_n; \Pi'_n)}{\sigma; (\Theta, \Theta', \Delta, \Delta') \vdash l \downarrow (V; \sigma'_n; \Pi'_n)} \quad (26)$$

$$\frac{l \notin \Delta \quad (l', l) \in \Theta \quad \sigma(l) = E \quad \sigma[l \mapsto \texttt{error}]; (\Theta \backslash \{(l', l)\}, \Theta' \cup \{(l', l)\}, \Delta, \Delta') \vdash l' \downarrow (V'; \sigma_2; \Pi_2) \quad \sigma_2[l \mapsto E]; \Pi_2 \vdash l \downarrow (V; \sigma_3; \Pi_3)}{\sigma; (\Theta, \Theta', \Delta, \Delta') \vdash l \downarrow (V; \sigma_2; \Pi_2)} \quad (27)$$

$$\frac{\Pi = (\Theta, \Theta', \Delta, \Delta') \quad l \notin \Delta \quad l \not\prec \Theta \quad \sigma(l) = E \quad \sigma[l \mapsto \texttt{error}]; \Pi \vdash E \downarrow (V; \sigma'; \Pi')}{\sigma; \Pi \vdash l \downarrow (V; \sigma'[l \mapsto V]; \Pi')} \quad (28)$$

$$\frac{\begin{array}{c}\sigma; \Pi \vdash M \downarrow (\langle \emptyset; o; x_i \overset{i \in I}{\mapsto} E_i; \pi \rangle; \sigma_2; \Pi_2) \quad \pi = (\theta, \delta) \quad \Pi_2 = (\Theta, \Theta', \Delta, \Delta') \\ \forall i \in I,\ l_i\ l'_i\ l''_i \text{ fresh } \ \Theta'' = \theta[l_i/x_i]_{i \in I}[l'_i/\hat{x_i}]_{i \in I}[l''_i/\overline{x_i}]_{i \in I} \quad \Delta'' = \delta[l_i/x_i]_{i \in I}[l'_i/\hat{x_i}]_{i \in I}[l''_i/\overline{x_i}]_{i \in I}\end{array}}{\sigma; \Pi \vdash \mathit{close}(M) \downarrow (\langle \emptyset; o; x_i \overset{i \in I}{\mapsto} l''_i; \mu(\{x_i \mid i \in I\}, \pi) \rangle; \sigma_2[l_i \overset{i \in I}{\mapsto} E_i[l'_i/x_i]_{i \in I}][l'_i \overset{i \in I}{\mapsto} l_i][l''_i \overset{i \in I}{\mapsto} l_i]; (\Theta \cup \Theta'', \Theta', \Delta \cup \Delta'', \Delta'))} \quad (29)$$

Figure 7. Evaluation rules for proving saneness of the semantics

A. Appendix

We extend a global constraint Π to a 4-tuple $(\Theta, \Theta', \Delta, \Delta')$ to accumulate eliminated constraints. Fig. 7 presents the necessary modifications. We let $dom(\Theta) = \{l \mid (l, l') \in \Theta\} \cup \{l' \mid (l, l') \in \Theta\}$ and $dom(\Delta) = \{l \mid l \in \Gamma, \Gamma \in \Delta\}$. $\Pi = (\Theta, \Theta', \Delta, \Delta')$ is *consistent* with σ if $dom(\Theta)$, $dom(\Theta')$, $dom(\Delta)$, $dom(\Delta') \subseteq dom(\sigma)$, and if $\sigma(l) = V$ then $l \notin \Pi$ and $l \not\prec \Theta$.

A binary relation $\prec_h$ on heap states is the smallest transitive relation satisfying the following conditions:

- $\sigma_1 \prec_h \sigma_2$ when $dom(\sigma_1) \subset dom(\sigma_2)$, and for all $l \in \sigma_1$, either $\sigma_1(l) = \sigma_2(l)$ or else $\sigma_1(l)$ is an expression but $\sigma_2(l)$ is not.
- $\sigma_1 \prec_h \sigma_2$ when $dom(\sigma_1) = dom(\sigma_2)$ and there is a non-empty subset $\mathcal{X}$ of $dom(\sigma_1)$ such that, for all $l \in \mathcal{X}$, $\sigma_1(l)$ is an expression but $\sigma_2(l)$ is not, and for all $l \in dom(\sigma_1) \backslash \mathcal{X}$, $\sigma_1(l) = \sigma_2(l)$.

We write $\sigma \preceq_h \sigma'$ when $\sigma \prec_h \sigma'$ or $\sigma = \sigma'$. Note that if there is an infinite sequence $\{\sigma_i\}_{i=0}^{\infty}$ such that for all i in 0, 1, 2, …, $\sigma_i \prec_h \sigma_{i+1}$, then we have an infinite sequence $\{dom(\sigma_i)\}_{i=0}^{\infty}$ where for all i in 0, 1, 2, …, $dom(\sigma_i) \subseteq dom(\sigma_{i+1})$, and for infinitely many i's, $dom(\sigma_j) \neq dom(\sigma_{j+1})$.

A binary relation $\prec_g$ on global constraints Π is the smallest transitive relation satisfying the following conditions:

- $(\Theta_1, \Theta'_1, \Delta_1, \Delta'_1) \prec_g (\Theta_2, \Theta'_2, \Delta_2, \Delta'_2)$ if $\Theta'_1 \subseteq \Theta'_2$ and $\Delta'_1 \subset \Delta'_2$.
- $(\Theta_1, \Theta'_1, \Delta_1, \Delta'_1) \prec_g (\Theta_2, \Theta'_2, \Delta_2, \Delta'_2)$ if $\Theta'_1 \subset \Theta'_2$ and $\Delta'_1 \subseteq \Delta'_2$.

A binary relation $\prec$ on triples of a heap state, global constraint and expression is the smallest transitive relation satisfying the following conditions:

- $(\sigma, \Pi, E) \prec (\sigma', \Pi', E')$ if $\sigma \prec_h \sigma'$ and $\Pi \preceq_g \Pi'$.
- $(\sigma, \Pi, E) \prec (\sigma', \Pi', E')$ if $\sigma \preceq_h \sigma'$ and $\Pi \prec_g \Pi'$.
- $(\sigma, \Pi, E) \prec (\sigma', \Pi', E')$ if $\sigma \preceq_h \sigma'$ and $\Pi \preceq_g \Pi'$ and E' is a sub-expression of E,. e.g. $E = \mathit{freeze}_{\psi}(E')$.
- $(\sigma, \Pi, M.X) \prec (\sigma', \Pi', l)$ if $\sigma \preceq_h \sigma'$ and $\Pi \preceq_g \Pi'$ and $l \in dom(\sigma')$.

LEMMA 1. *If there is an infinite sequence $\{(\sigma_i, \Pi_i, E_i)\}_{i=1}^{\infty}$ such that for all i in $1, 2, \dots$, $(\sigma_i, \Pi_i, E_i) \prec (\sigma_{i+1}, \Pi_{i+1}, E_{i+1})$ and Π_i is consistent with σ_i, then for all i in $1, 2, \dots$, $\sigma_i \preceq_h \sigma_{i+1}$ and for infinitely many i's, $\sigma_i \prec_h \sigma_{i+1}$.*

LEMMA 2. *If $\sigma; \Pi \vdash E \downarrow (V; \sigma'; \Pi')$ and Π is consistent with σ, then $\sigma \preceq_h \sigma'$, $\Pi \preceq_g \Pi'$, Π' is consistent with σ', and if $E = l$ then $\sigma'(l) = V$.*

Proof. By induction on the derivation of $\sigma; \Pi \vdash E \downarrow (V; \sigma'; \Pi')$. □

We prove saneness of the semantics for well-typed terms. Precisely, we assume projections are only made from closed mixins; thus in the second hypothesis of rule (18), we have $\rho \circ o(X) = l$ for some l. Mal-formed terms can make the semantics insane, i.e. divergence without heap being extended.

LEMMA 3. *If $\sigma_1; \Pi_1 \vdash E_1 \downarrow (V_1; \sigma'_1; \Pi'_1)$ is an immediate hypothesis of $\sigma_2; \Pi_2 \vdash E_2 \downarrow (V_2; \sigma'_2; \Pi'_2)$, then $(E_2, \sigma_2, \Pi_2) \prec (E_1, \sigma_1, \Pi_1)$.*

Proof. By case on the last rule used and Lemma 2. □

Then Lemma 1 and Lemma 3 together deduce if the derivation for $\emptyset; (\emptyset, \emptyset) \vdash E \downarrow (\_; \_; \_)$ grows infinitely, then the heap is extended infinitely.