Rigid Mixin Modules

Tom Hirschowitz

Rigid Mixin Modules

Tom Hirschowitz

ENS Lyon

Abstract. Mixin modules are a notion of modules that allows cross-module recursion and late binding, two features missing in ML-style modules. They have been well defined in a call-by-name setting, but in a call-by-value setting, they tend to conflict with the usual static restrictions on recursive definitions. Moreover, the semantics of instantiation has to specify an order of evaluation, which involves a difficult design choice. Previous proposals [14, 16] rely on the dependencies between components to compute a valid order of evaluation. In such systems, mixin module types must carry some information on the dependencies between their components, which makes them verbose. In this paper, we propose a new, simpler design for mixin modules in a call-by-value setting, which avoids this problem.

1 Introduction

1.1 The problem

For programming "in the large", it is desirable that the programming language offers linguistic support for the decomposition and structuring of programs into modules. A good example of such linguistic support is the ML module system and its powerful notion of parameterized modules. Nevertheless, this system is weak on two important points.

(Mutual recursion) Mutually recursive definitions cannot be split across separate modules. There are several cases where this hinders modularization [6].
(Modifiability) The language does not propose any mechanism for incremental modification of an already-defined module, similar to inheritance and overriding in object-oriented languages.

Class-based object-oriented languages provide excellent support for these two features. Classes are naturally mutually recursive, and inheritance and method overriding answer the need for modifiability. However, viewed as a module system, classes have two weaknesses: they do not offer a general parameterization mechanism (no higher-order functions on classes), and the mechanisms they offer to describe pre-computations (initialization of static and instance variables) lack generality, since a module system should allow to naturally alternate function definitions with computational definitions using these functions.

Mixin modules [4] (hereafter simply called "mixins") provide an alternative approach to modularity that combines some of the best aspects of classes and

ML-style modules. Mixins are modules with "holes" (not-yet-defined components), where the holes can be plugged later by composition with other mixins, following a late-binding semantics. However, the handling of pre-computations and initializations in mixins is still problematic. Most of the previous work on mixins, notably by Ancona and Zucca [2] and Wells and Vestergaard [20], is better suited to a call-by-name evaluation strategy. This strategy makes it impossible to trigger computations at initialization time (see Sect. 6 for more details).

The choice of a call-by-value setting raises the following two issues.

(Recursive definitions) Since mixin components are not necessarily functions, arbitrary recursive definitions can appear dynamically by composition. For instance, consider the following two mixins, (in an informal concrete syntax)

```
mixin A = mix                 mixin B = mix
  ? x : int                     ? y : int
  ! y = x + 1        and        ! x = y * 2
end                           end
```

Each of these mixins declares the missing value (marking it with `?`) and defines the other one (marking it with `!`). The composition of `A` and `B` involves the mutually recursive definition `x = y * 2` **and** `y = x + 1`.
In most call-by-value languages, recursive definitions are statically restricted, in order to be more efficiently implementable [3, 15], and to avoid some ill-founded definitions. Obviously, our system should not force language designers to abandon these properties, and thus needs guards on recursive definitions, at the level of both static and dynamic semantics.

(Order of evaluation) In our system, mixins will contain arbitrary, unevaluated definitions, whose evaluation will be triggered by instantiation. Because these definitions are arbitrary, the order in which they will be evaluated matters. For instance, in a mixin `A` defining `x = 0` and `y = x + 1`, `x` must be evaluated before `y`. Thus, the semantics of instantiation must define an order of evaluation. Moreover, mixins can be built by composition, so the semantics of composition must also take the order of definitions into account.

From the standpoint of dynamic semantics, the second issue involves a design decision. From the standpoint of typing, it reduces to the first issue, since the existence of a valid order of evaluation is governed by the absence of invalid recursive definitions.

1.2 Instantiation-time ordering: flexible mixin modules

The *MM* language of call-by-value mixins [14, 16, 12] is designed as follows. Mixins contain unordered definitions. Only at instantiation does the system compute an order for them, according to their inter-dependencies [14, 16, 12], and to programmer-supplied annotations that fix some bits of the final order [16, 12]. This solution, which we call *flexible mixins* is very expressive w.r.t. code reuse, since components can be re-ordered according to the context. However, it appears too complex in some respects.

(Instantiation) In particular, instantiation is too costly, since it involves computing the strongly-connected components of a graph whose size is quadratic in the input term, plus a topological sort of the result.

(Type safety) As explained above, when recursive definitions are restricted, the type system must prevent invalid ones. In *MM*, mixin types contain some information about the dependencies between definitions. Nevertheless, this makes mixin types verbose, and also over-specified, in the sense that any change in the dependencies between components can force the type of the mixin to change, which is undesirable.

The first problem is not so annoying in the context of a module system: it only has to do with linking operations, and thus should not affect the overall efficiency of programs. The second problem makes the proposed language impractical without dedicated graph support.

1.3 Early ordering: rigid mixin modules

In this paper, we propose a completely different approach, from scratch. We introduce *Mix*, a new language of call-by-value mixins, where mixin components are ordered, in a rigid way. They can be defined either as single components (briefly called "singles") or as blocks of components. Blocks contain mutually recursive definitions, and are restricted to a certain class of values. Conversely, singles can contain arbitrary, non-recursive computations. Composition preserves the order of both of its arguments, and instantiation straightforwardly converts its argument into a module.

In *Mix*, the components of a mixin are ordered once for all at definition time, so *Mix* is less expressive than *MM*. Yet, it has other advantages. First, with respect to side effects, annotations are no longer needed, since side effects always respect the syntactic order. Moreover, instantiation is less costly than in *MM*, since it runs in $O(n \ log \ n)$, where n is the size of the input. Concerning typing, mixin types have the same structure as mixins themselves: they are sequences of specifications, which can be either singles or blocks. They avoid the use of explicit graphs, which improves over *MM*. Compared to ML module types, the only differences are that the order matters and that mutually recursive specifications must be explicitly grouped together. Finally, the meta theory of *Mix* is much simpler than the one of *MM*, which makes it more likely to scale up to a fully-featured language. In summary, we propose *Mix* as a good trade-off between expressiveness and user-friendliness for incorporating mixins into a practical programming language like ML.

The rest of the paper is organized as follows. Section 2 presents an informal overview of *Mix* by example. Section 3 formally defines *Mix* and its dynamic semantics. Section 4 defines a sound type system for *Mix*. Finally, sections 5 and 6 review related and future work, respectively. The proofs are omitted from this paper for lack of space, but a longer version including them is available as a research report [13].

2 Intuitions

As a simplistic introductory example, consider a program that defines two mutually recursive functions for testing whether an integer is even or odd, and then tests whether 56 is even, and whether it is odd. Assume now that it is conceptually obvious that everything concerning oddity must go into one program fragment, and everything concerning evenness must go into another, clearly distinct fragment. Here is how this can be done in an informal programming language based on *Mix*, with a syntax mimicking OCaml [17].

First, define two mixins `Even` and `Odd` as follows.

```
mixin Even = mix
  recblock ? odd : int -> bool
       and  ! even x = x = 0 or odd (x-1)
  ! even56 = even 56
end

mixin Odd = mix
  recblock ? even : int -> bool
        and ! odd x = x > 0 and even (x-1)
  ! odd56 = odd 56
end
```

Each of these mixins declares the missing function (marking it with `?`) and defines the other one (marking it with `!`), inside a **`recblock`** which delimits a recursive block. Then, outside of this block, each mixin performs one computation.

In order to link them, and obtain the desired complete mixin, one composes `Even` and `Odd`, by writing **`mixin`** `OpenNat = Odd >> Even`. Intuitively, composition connects the missing components of `Even` to the corresponding definitions of `Odd`, and *vice versa*, preserving the order of both mixins. Technically, composition somehow passes `Odd` through `Even`, with `Even` acting as a filter, stopping the components of `Odd` when they match one of its own components. This filtering is governed by some rules: the components of `Odd` go through `Even` together, until one of them, say component c, matches some component of `Even`. Then, the components of `Odd` defined to the left of c are stuck at the current point. The other components continue their way through `Even`. Additionally, when two components match, they are merged into a single component.

In our example, `odd` and `even` both stop at `Even`'s recursive block mentioning them, so the two recursive blocks are merged. Further, `odd56` remains is unmatched, so it continues until the end of `Even`. The obtained mixin is thus equivalent to

```
mixin OpenNat = mix
  recblock ! even x = x = 0 or odd (x-1)
        and ! odd x = x > 0 and even (x-1)
  ! even56 = even 56
  ! odd56 = odd 56
end
```

Note that composition is asymmetric. This mixin remains yet to be instantiated, in order to trigger its evaluation. This is done by writing **module** `Nat =` **close** `OpenNat`, which makes `OpenNat` into a module equivalent to

```
module Nat = struct
  let rec even x = x = 0 or odd (x-1)
          odd x = x > 0 and even (x-1)
  let even56 = even 56
  let odd56 = odd 56
end
```

which evaluates to the desired result. For comparison, in *MM*, the final evaluation order would be computed upon instantiation, instead of composition. It would involve a topological sort of the strongly-connected components of the dependency graph of `OpenNat`. Incidentally, in *MM*, in order to ensure that `even56` is evaluated before `odd56`, the definition of `odd56` should better explicitly state it. From the standpoint of typing, explicitly grouping possibly recursive definitions together allows to get rid of dependency graphs in the types, thus greatly simplifying the type system.

3 The *Mix* language and its dynamic semantics

3.1 Syntax

Pre-terms. Figure 1 defines the set of *pre-terms* of *Mix*. It distinguishes names X from variables x, following Harper and Lillibridge [11]. It includes a standard record construct $\{s\}$, where $s ::= (X_1 = e_1 \dots X_n = e_n)$, and selection $e.X$. It features two constructs for value binding, letrec for mutually recursive definitions, and let for single, non-recursive definitions. Finally, the language provides four mixin constructs. Basic mixins consist of *structures* $m = (c_1 \dots c_n)$, wich are lists of *components*. A component c is either a *single*, or a *block*. A single u is either a named *declaration* $X \rhd x = \bullet$, or a *definition* $L \rhd x = e$, where L is a *label*. Labels can be names or the special *anonymous* label, written $\_$, in which case the definition is also said anonymous. Finally, a block q is a list of singles. The other constructs are composition $(e_1 \gg e_2)$, instantiation $(\mathsf{close}\ e)$, and deletion of a name X, written $(e_{|-X})$.

Terms. Proper terms are defined by restricting the set of pre-terms, as follows. We define *Mix values* v by $v ::= \{s^v\} \mid \langle m \rangle$, where $s^v ::= (X_1 = v_1 \dots X_n = v_n)$. Then The system is parameterized by the set *RecExp* of *valid recursive expressions*, which must contain only values, and be closed under substitution.

Definition 1 (Terms)
A term *of Mix is a pre-term such that: records do not define the same name twice ; bindings do not define the same variable twice ; structures define neither*

x	$\in$	*Vars*	Variable
X	$\in$	*Names*	Name
L	$\in$	$\mathit{Names} \cup \{\_\}$	Label
Expression:			
e	$::=$	x	Variable
	$\mid$	$\{X_1 = e_1 \dots X_n = e_n\}$	Record
	$\mid$	$e.X$	Selection
	$\mid$	letrec $x_1 = e_1 \dots x_n = e_n$in e	letrec
	$\mid$	let $x = e_1$ in e_2	let
	$\mid$	$\langle c_1 \dots c_n \rangle$	Structure
	$\mid$	$e_1 \gg e_2$	Composition
	$\mid$	close e	Instantiation
	$\mid$	$e_{\mid -X}$	Deletion
Definition:			
c	$::=$	u	Single definition
	$\mid$	$[u_1 \dots u_n]$	Block
Single definition:			
u	$::=$	$L \rhd x = e \mid X \rhd x = \bullet$	

Fig. 1. Syntax

the same name twice nor the same variable twice ; and the right-hand sides of letrec *and block definitions belong to RecExp.*

The restriction of letrec and block definitions to valid recursive expressions both simplifies the semantics of letrec, and models the restrictions put by standard call-by-value languages on recursive definitions [17, 19]. Typically, recursive definitions can only be functions.

Records, bindings and structures are respectively considered as finite maps from names to terms, variables to terms, and pairs of a label and a variable to terms or $\bullet$. Thus, given a structure m, the restriction of $dom(m)$ (the domain of m) to pairs of a name and a variable can be seen as an injective finite map from names to variables, which we call $\mathit{VofN}(m)$.

Terms are considered equivalent modulo proper [20] renaming of bound variables and modulo the order in blocks and letrec. We denote by $DV(m)$ and $DV(b)$ the sets of variables defined by m and b, respectively, and by $DN(m)$ and $DN(s)$ the sets of names defined by m and s, respectively.

3.2 Dynamic semantics

The semantics of *Mix* is defined as a reduction relation on pre-terms in figure 2, using notions defined in figure 3. It is compatible with bound variable renaming and preserves well-formedness, so we extend it to terms.

Figure 3 defines *evaluation contexts*, which enforce a deterministic, call-by-value strategy. We can now examine the rules, from the most interesting to the most standard.

$$
\begin{array}{rll}
\langle m_1 \rangle \gg \langle m_2 \rangle \to \langle \mathit{Add}(m_1, m_2, \varepsilon) \rangle \text{ if } m_1 \Bumpeq m_2 & & (\textsc{Compose}) \\
\mathsf{close}\ \langle m^c \rangle \to \mathit{Bind}(m^c, \{\mathit{Record}(m^c)\}) & & (\textsc{Close}) \\
\langle m \rangle_{|-X} \to \langle \mathit{Del}(m, X) \rangle & & (\textsc{Delete}) \\
\mathsf{letrec}\ b\ \mathsf{in}\ e \to \{x \mapsto \mathsf{letrec}\ b\ \mathsf{in}\ b(x) \mid x \in \mathit{dom}(b)\}(e) & & (\textsc{LetRec}) \\
\mathsf{let}\ x = v\ \mathsf{in}\ e \to \{x \mapsto v\}(e) & & (\textsc{Let}) \\
\{s^v\}.X \to s^v(X) & & (\textsc{Select}) \\
\mathbb{E}[e] \to \mathbb{E}[e'] \text{ if } e \to e' & & (\textsc{Context})
\end{array}
$$

Fig. 2. Reduction rules

Composition. Rule COMPOSE describes mixin composition. In order to be composed, the structures must be made *compatible* by α-conversion. Namely, we say that two structures m_1 and m_2 are compatible, and write $m_1 \Bumpeq m_2$, iff $DV(m_1) \cap FV(\langle m_2 \rangle) = DV(m_2) \cap FV(\langle m_1 \rangle) = \emptyset$, and for any $x \in DV(m_1) \cap DV(m_2)$, there exists a name X such that $\mathit{VofN}(m_1)(X) = \mathit{VofN}(m_2)(X) = x$. This basically says that both structures agree on the names of variables.

Then, their composition $\langle m_1 \rangle \gg \langle m_2 \rangle$ is $\mathit{Add}(m_1, m_2, \varepsilon)$, where Add is defined by induction on m_1 by

$$
\begin{array}{rcl}
\mathit{Add}(\varepsilon, m_1, m_2) & = & m_1, m_2 \\
\mathit{Add}((m_1, c), m_2, m_3) & = & \mathit{Add}(m_1, m_2, (c, m_3)) \\
& & \quad \text{if } \mathit{DN}(c) \cap \mathit{DN}(m_2, m_3) = \emptyset \\
\mathit{Add}((m_1, c_1), (m_2^1, c_2, m_2^2), m_3) & = & \mathit{Add}(m_1, m_2^1, (c_1 \otimes c_2, m_2^2, m_3)) \\
\multicolumn{3}{r}{\text{if } \mathit{DN}(c_1) \cap \mathit{DN}(m_2^1, m_2^2, m_3) = \emptyset \text{ and } \mathit{DN}(c_1) \cap \mathit{DN}(c_2) \neq \emptyset}
\end{array}
$$

Given three arguments m_1, m_2, m_3, Add roughly works as follows. If m_1 is empty, it returns the concatenation of m_2 and m_3. If the last component c of m_1 defines names that are not defined in m_2 or m_3, then c is pushed at the head of m_3. Finally, when the last component c_1 of m_1 defines a name also defined by some c_2 in m_2, so that $m_2 = (m_2^1, c_2, m_2^2)$, then the third argument becomes $(c_1 \otimes c_2, m_2^2, m_3)$, where $c_1 \otimes c_2$ is the *merging* of c_1 and c_2, which is defined by

$$
\begin{array}{rcll}
c_1 \otimes c_2 & = & c_2 \otimes c_1 & \\
(X \rhd x = \bullet) \otimes c & = & c & \text{if } \mathit{VofN}(c)(X) = x \\
[q_1] \otimes [q_2] & = & [q_1, q_2] & \text{if } \mathit{DN}(q_1) \cap \mathit{DN}(q_2) = \emptyset \\
[X \rhd x = \bullet, q_1] \otimes [q_2] & = & [q_1] \otimes [q_2] & \text{if } \mathit{VofN}(q_2)(X) = x
\end{array}
$$

This definition is not algorithmic, but uniquely defines the merging of two components, and an algorithm is easy to derive from it: one has to apply rules 2 and 4 as long as possible, then commute the arguments and apply rules 2 and 4 as long as possible again, and then finally apply rule 3. Technically, as soon as a declaration is matched, it is removed, and when two blocks have no more common defined names, their merging is their union. Note that initially, only components with common defined names are merged, but that the union takes place after all the common names have been reduced.

$$\begin{aligned}\mathbb{E} ::= {} & \{s^v, X = \Box, s\} \mid \Box.X \\ & \mid \mathsf{let}\ x = \Box\ \mathsf{in}\ e \\ & \mid \Box \gg e \mid v \gg \Box \\ & \mid \mathsf{close}\ \Box \mid \Box_{|-X}\end{aligned}$$

Fig. 3. Evaluation contexts

Example 1. Assuming that *Mix* is extended with functions, integers and booleans, the mixin `Even` described in section 2 is written

$$\begin{aligned} \mathit{even} = \langle\, [\, & \mathit{Odd} \rhd \mathit{odd} = \bullet, \\ & \mathit{Even} \rhd \mathit{even} = \lambda x.(x = 0)\ \mathsf{or}\ \mathit{odd}\ (x-1)\ ], \\ & \mathit{Even56} \rhd \mathit{even56} = \mathit{even}\ 56\ \rangle \end{aligned}$$

During composition with the mixin corresponding to `Odd`, the component *Odd56* traverses the whole structure to go to the rightmost position, then the two blocks defining *Even* and *Odd* are merged, which gives the expected block.

Instantiation. Rule Close describes the instantiation of a *complete* basic mixin. A structure is said *complete* iff it does not contain declarations. We denote complete structures, components, singles, and blocks by m^c, c^c, u^c, and q^c, respectively. Given a complete basic mixin $\langle m^c \rangle$, instantiation first generates a series of bindings, following the structure of m^c, and then stores the results of named definitions in a record. Technically, $\mathsf{close}\ \langle m^c \rangle$ reduces to $\mathit{Bind}(m^c, \{\mathit{Record}(m^c)\})$, where *Record* makes m^c into a record and *Bind* makes m^c into a binding:

$\mathit{Record}(m^c)$ is defined on singles by

$$\mathit{Record}(X \rhd x = e) = (X = x) \qquad \text{and} \qquad \mathit{Record}(\_ \rhd x = e) = \varepsilon,$$

naturally extended to components and structures by concatenation,

and $\mathit{Bind}(m^c, e)$ is defined inductively over m^c by

$$\begin{aligned} \mathit{Bind}(\varepsilon, e) &= e \\ \mathit{Bind}(([u^c{}_1 \dots u^c{}_n], m^c), e) &= \mathsf{letrec}\ \lfloor u^c{}_1 \rfloor \dots \lfloor u^c{}_n \rfloor\ \mathsf{in}\ \mathit{Bind}(m^c, e) \\ \mathit{Bind}((u^c, m^c), e) &= \mathsf{let}\ \lfloor u^c \rfloor\ \mathsf{in}\ \mathit{Bind}(m^c, e), \\ \text{with } \lfloor L \rhd x = e \rfloor &= (x = e). \end{aligned}$$

For each component, *Bind* defines a letrec (if the component is a block) or a let (if the component is a single), by extracting bindings $x = e$ from singles $L \rhd (x = e)$.

Other rules. Rule Delete describes the action of the deletion operation. Given a basic mixin $\langle m \rangle$, $\langle m \rangle_{|-X}$ reduces to $\langle \mathit{Del}(m, X) \rangle$, where $\mathit{Del}(m, X)$ denotes m, where any definition of the shape $X \rhd x = e$ is replaced with $X \rhd x = \bullet$.

Type:

$$\begin{aligned}
\tau \in \mathit{Types} &::= \{S\} \mid \langle C_1 \dots C_n \rangle \\
C &::= U \mid [U_1 \dots U_n] \\
U &::= \delta X : \tau \\
\delta &::= \,! \mid ? \\
S &\in \mathit{Names} \xrightarrow{\mathit{fin}} \mathit{Types}
\end{aligned}$$

Environment:

$$\Gamma \in \mathit{Vars} \xrightarrow{\mathit{fin}} \mathit{Types}$$

Fig. 4. Types

The next two rules, LetRec, Let, handle value binding. The only non-obvious rule is LetRec, which enforces the following behavior. The idea is that the rule applies when the considered binding is fully evaluated, which is always the case for proper terms. A pre-term letrec b in e reduces to e, where each $x \in \mathit{dom}(b)$ is replaced with a kind of closure representing its definition, namely letrec b in $b(x)$. Note the notation for capture-avoiding substitution.

Finally, rule Select defines record selection, and rule Context extends the rule to any evaluation context.

4 Static semantics

We now define a sound type system for *Mix* terms. Defining it on terms rather than pre-terms means that the considered expressions are well-formed by definitions. Types are defined in figure 4. A *Mix* type τ can be either a record type or a mixin type. A mixin type has the shape $\langle M \rangle$, where M is a *signature.* A signature is a list of *specifications* C, which can be either *single specifications* U or *block specifications* Q. A single specification has the shape $\delta X : \tau$ where δ is a flag indicating whether the considered name is a declaration of a definition. It can be either ?, for declarations, or !, for definitions. A block specification is a list of single specifications. Record types are finite maps from names to types. Types are identified modulo the order of specifications in blocks. Environments Γ are finite maps from variables to types. The disjoint union of two environments Γ_1 and Γ_2 is written $\Gamma_1 + \Gamma_2$ (which applies only if their domains are disjoint).

Figure 5 presents our type system for *Mix*.

Basic mixin modules and enriched specifications. Let us begin with the typing of basic mixins. Rule T-Struct simply delegates the typing of a basic mixin $\langle m \rangle$ to the rules for typing structures. These rules basically give each component c an *enriched specification*, which is a specification, enriched with the corresponding variable. Formally, single enriched specifications have the shape $\delta L \rhd x : \tau$, and enriched block specifications are finite sets of these. Notably, this allows to type anonymous definitions (using enriched specifications like $\delta_{\_} \rhd x : \tau$), and also to recover a typing environment (namely $\{x \mapsto \tau\}$) for typing the next components.

Expressions

T-Struct
$$\frac{\Gamma \vdash c_1 \ldots c_n : M^e}{\Gamma \vdash \langle c_1 \ldots c_n \rangle : \langle \mathit{Sig}(M^e) \rangle}$$

T-Compose
$$\frac{\Gamma \vdash e_1 : \langle M_1 \rangle \qquad \Gamma \vdash e_2 : \langle M_2 \rangle}{\Gamma \vdash e_1 \gg e_2 : \langle \mathit{Add}(M_1, M_2, \varepsilon) \rangle}$$

T-Close
$$\frac{\Gamma \vdash e : \langle M^c \rangle}{\Gamma \vdash \mathsf{close}\ e : \{\mathit{Record}(M^c)\}}$$

T-Delete
$$\frac{\Gamma \vdash \langle m \rangle : \langle M \rangle}{\Gamma \vdash \langle m \rangle_{|-X} : \langle \mathit{Del}(M, X) \rangle}$$

T-Var
$$\Gamma \vdash x : \Gamma(x)$$

T-Record
$$\frac{\mathit{dom}(s) = \mathit{dom}(S) \qquad \forall X \in \mathit{dom}(s), \Gamma \vdash s(X) : S(X)}{\Gamma \vdash \{s\} : \{S\}}$$

T-Select
$$\frac{\Gamma \vdash e : \{S\}}{\Gamma \vdash e.X : S(X)}$$

T-LetRec
$$\frac{\Gamma + \Gamma_b \vdash b : \Gamma_b \qquad \Gamma + \Gamma_b \vdash e : \tau}{\Gamma \vdash \mathsf{letrec}\ b\ \mathsf{in}\ e : \tau}$$

T-Let
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma + \{x \mapsto \tau_1\} \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Singles

T-Some
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (L \rhd x = e) : (!L \rhd x : \tau)}$$

T-None
$$\Gamma \vdash (X \rhd x = \bullet) : (?X \rhd x : \tau)$$

Structures and bindings

T-Empty
$$\Gamma \vdash \varepsilon : \varepsilon$$

T-Single
$$\frac{\Gamma \vdash u : U^e \qquad \Gamma + \mathit{Env}(U^e) \vdash m : M^e}{\Gamma \vdash (u, m) : (U^e, M^e)}$$

T-Block
$$\frac{\begin{array}{c}\Gamma + \Gamma_q \vdash m : M^e \qquad \Gamma_q = \biguplus_{u \in q} \mathit{Env}({U^e}_u) \\ \forall u \in q, u \in \mathit{RecExp}_? \text{ and } \Gamma + \Gamma_q \vdash u : {U^e}_u\end{array}}{\Gamma \vdash ([q], m) : ([\biguplus_{u \in q} {U^e}_u], M^e)}$$

T-Binding
$$\frac{\begin{array}{c}\mathit{dom}(b) = \mathit{dom}(\Gamma_b) \\ \forall x \in \mathit{dom}(b), b(x) \in \mathit{RecExp} \text{ and } \Gamma_b \vdash b(x) : \Gamma_b(x)\end{array}}{\Gamma \vdash b : \Gamma_b}$$

Fig. 5. Type system

Enriched single specifications, block specifications, and signatures are denoted by U^e, Q^e, and M^e, respectively. Once the structure m has been given such an enriched signature M^e, this result is converted to a proper signature $M = \mathit{Sig}(M^e)$, assigning to the basic mixin $\langle m \rangle$ the type $\langle M \rangle$. The *Sig* function merely forgets variables and anonymous definitions of its argument: it is defined by straightforward extension of

$$\mathit{Sig}(\delta X \rhd x : \tau) = \delta X : \tau \qquad \mathit{Sig}(\delta_\_ \rhd x : \tau) = \varepsilon.$$

Here is how structures are given such enriched signatures. By rule T-Some, a single definition $L \rhd x = e$ is given the enriched single specification $!L \rhd x : \tau$ if e has type τ. By rule T-None, a single declaration $X \rhd x = \bullet$ can be given any enriched specification of the shape $?X \rhd x : \tau$.

Given this, we can define the typing of structures. By rule T-Empty, an empty structure is given the empty signature. By rule T-Single, a structure of the shape (u, m) is typed as follows. First, u is typed, yielding an enriched specification U^e. This U^e is made into an environment by the *Env* function from enriched signatures to environments. This function associates to any enriched single specification $\delta L \rhd x : \tau$ the finite map $\{x \mapsto \tau\}$, and is straightforwardly extended to signatures. The obtained environment is added to the current environment for typing the remaining components inductively, yielding an enriched signature M^e. The type of the whole structure is (U^e, M^e).

By rule T-Block, a structure of the shape $([q], m)$ is typed as follows. An enriched single specification ${U^e}_u$ is guessed for each single u of q. Then, the set of these enriched single specifications is converted into an environment $\Gamma_q = \biguplus_{u \in q} \mathit{Env}({U^e}_u)$. This environment Γ_q is added to the current environment. Then, it is checked that each single u indeed has the enriched specification ${U^e}_u$. Additionally, it is checked that each single u of q is defined by a valid recursive expression or is a declaration. By abuse of notation, we write this $u \in \mathit{RecExp}_?$. Finally, the structure m is typed, yielding an enriched signature M^e, which is concatenated to $[\biguplus_{u \in q} {U^e}_u]$.

Composition. The typing of composition, defined by rule T-Compose, recalls its dynamic semantics. The type of the composition of two mixins of types $\langle M_1 \rangle$ and $\langle M_2 \rangle$, respectively, is $\langle \mathit{Add}(M_1, M_2, \varepsilon) \rangle$, where *Add* is defined by

$$\begin{array}{rcl}
\mathit{Add}(\varepsilon, M_1, M_2) & = & M_1, M_2 \\
\mathit{Add}((M_1, C), M_2, M_3) & = & \mathit{Add}(M_1, M_2, (C, M_3)) \\
& & \quad \text{if } \mathit{DN}(C) \cap \mathit{DN}(M_2, M_3) = \emptyset \\
\mathit{Add}((M_1, C_1), (M_2^1, C_2, M_2^2), M_3) & = & \mathit{Add}(M_1, M_2^1, (C_1 \otimes C_2, M_2^2, M_3)) \\
\multicolumn{3}{r}{\text{if } \mathit{DN}(C_1) \cap \mathit{DN}(M_2^1, M_2^2, M_3) = \emptyset \text{ and } \mathit{DN}(C_1) \cap \mathit{DN}(C_2) \neq \emptyset}
\end{array}$$

which does the same as *Add* on structures. The merging of two specifications is similarly defined by

$$\begin{array}{rcll} C_1 \otimes C_2 & = & C_2 \otimes C_1 & \\ (?X : \tau) \otimes C & = & C & \text{if } C(X) = \tau \\ [Q_1] \otimes [Q_2] & = & [Q_1, Q_2] & \text{if } \mathit{DN}(Q_1) \cap \mathit{DN}(Q_2) = \emptyset \\ [?X : \tau, Q_1] \otimes [Q_2] & = & [Q_1] \otimes [Q_2] & \text{if } Q_2(X) = \tau \end{array}$$

It differs from component merging, because it checks that the types of matching specifications are the same.

Example 2. When some mutually recursive definitions are grouped together in blocks, rule T-Block ensures that they are all defined by valid recursive expressions. Let us now show how the type system rules out mutually recursive definitions that are not in blocks. Assume given mixins with types $e_1 : \langle ?Y : \tau_Y, !X : \tau_X \rangle$ and $e_2 : \langle ?X : \tau_X, !Y : \tau_Y \rangle$. When typing their composition $e_1 \gg e_2$, the component X in e_1, by the third rule in the definition of *Add*, is merged with its counterpart in e_2. This pushes the component Y of e_2 to the right, so that we obtain the triple $(?Y : \tau_Y, \varepsilon, (!X : \tau_X, !Y : \tau_Y))$, to which no rule applies.

Other rules. Rule T-Close types instantiation. Following previous notation, we let M^c denote complete signatures. Given a complete mixin of type $\langle M^c \rangle$, close makes it into a record. The type of the result is $\{\mathit{Record}(M^c)\}$, which is obtained by flattening the blocks in M^c, forgetting the ! flags.

Rule T-Delete types deletion. For a mixin e of type $\langle M \rangle$, the rule gives $e_{|-X}$ the type $\langle \mathit{Del}(M, X) \rangle$, in which $\mathit{Del}(M, X)$ denotes M, where any declaration of the shape $!X : \tau$ is replaced with $?X : \tau$.

The other typing rules are straightforward.

Soundness. The type sytem is sound, in the sense that the following results hold.

Lemma 1 (Subject reduction)
If $\Gamma \vdash e : \tau$ and $e \rightarrow e'$, then $\Gamma \vdash e' : \tau$.

Lemma 2 (Progress)
If $\emptyset \vdash e : \tau$, then either e is a value, or there exists e' such that $e \rightarrow e'$.

Theorem 1 (Soundness)
If $\emptyset \vdash e : \tau$, then either e reduces to a value, or its evaluation does not terminate.

5 Related work

Kernel calculi with mixin modules. The idea of mixin modules comes from that of mixins, introduced by Bracha [4] as a model of inheritance. In this model, called Jigsaw, classes are represented by *mixins*, which are equipped with a powerful set of modularity operations, and can be instantiated into *objects*. Syntactically, mixins may contain only values, which makes them as restrictive as classes. What differentiates them from classes is their cleaner design, which gave other authors the idea to generalize them to handle modules as well as objects.

Ancona and Zucca [2] propose a call-by-name module system based on some of Bracha's ideas, called *CMS*. As *Mix*, *CMS* extends Jigsaw by allowing any kind of expressions as mixin definitions, not just values. Unlike in *Mix*, in *CMS*, there is no distinction between modules and mixin modules, which makes sense in call-by-name languages, since the contents of modules are not evaluated until selection. In call-by-value, the contents of a module are eagerly evaluated, so they cannot have a late binding semantics. Thus, modules must be dinstinguished from mixin modules, and so *CMS* is not a suitable model. From the standpoint of typing, *CMS*, unlike *Mix*, but consistently with most call-by-name languages, does not control recursive definitions.

The separation between mixin modules and modules, as well as late binding, can be encoded in Wells and Vestergaard's **m**-calculus [20], which however is untyped, and does not provide programmer control over the order of evaluation.

In a more recent calculus [1], Ancona et al. separate mixin modules from modules, and handle side effects as a monad. However, they do not attempt to statically reject faulty recursive definitions. Moreover, in their system, given a composition $e_1 \gg e_2$, the monadic (i.e., side-effective) definitions of e_1 are necessarily evaluated before those of e_2, which is less flexible than our proposal.

Language designs with mixin modules. Duggan and Sourelis [8] propose an extension of ML with mixin modules, where mixin modules are divided into a *prelude*, a *body*, and an *initialization section*. Only definitions from the body are concerned by mixin module composition, the other sections being simply concatenated (and disjoint). Also, the body is restricted to functions and data-type definitions, which prevents illegal recursive definitions from arising dynamically. This is less flexible than *Mix*, since it considerably limits the alternation of functional and computational definitions.

Flatt and Felleisen [10] introduce the closely related notion of *units*, in the form of (1) a theoretical extension to Scheme and ML and (2) an actual extension of their PLT Scheme implementation of Scheme [9]. In their theoretical work, they only permit values as unit components, except for a separate initialization section. This is more restrictive than *Mix*, in the same way as Duggan and Sourelis. In the implementation, however, the semantics is different. Any expression is allowed as a definition, and instantiation works in two phases. First, all fields are initialized to `nil`; and second, they are evaluated and updated, one after another. This yields both unexpected behavior (consider the definition `x = cons(1, x)`), and dynamic type errors (consider `x = x + 1`), which do not occur in *Mix*. Finally, units do not feature late binding, contrarily to *Mix*.

Linking calculi. Other languages that are close to mixin modules are linking calculi [5, 18]. Generally, they support neither nested modules nor late binding, which significantly departs from *Mix*. Furthermore, among them, Cardelli's proposal [5] does not restrict recursion at all, but the operational semantics is sequential in nature and does not appear to handle cross-unit recursion. As a result, the system seems to lack the progress property. Finally, Machkasova and

Turbak [18] explore a linking calculus with a very rich equational theory, but which does not restrict recursion and is untyped.

Flexible mixin modules. In the latest versions of *MM* [12], a solution to the problem of dependency graphs in types is proposed. Instead of imposing that the graph in a mixin module type exactly reflect the dependencies of the considered mixin module, it is seen as a bound on its dependencies, thanks to an adequate notion of subtyping. Roughly, it ensures that the considered mixin module has no more dependencies than exposed by the graph. This allows two techniques for preventing *MM* mixin module types from being verbose and over-specified. First, the interfaces of a mixin module e can be given more constrained dependency graphs than that of e. This makes interfaces more robust to later changes. Second, a certain class of dependency graphs is characterized, that bear a convenient syntactic description, thus avoiding users to explicitly write graphs by hand. In fact, this syntactic sugar allows to write *MM* types exactly as *Mix* types. We call MM^2 the language obtained by restricting *MM* to such types (in a way that remains to be made precise, for instance by insertion of implicit coercions).

The expressive power of typed MM^2 w.r.t. reordering lies between *MM* and *Mix*. Intuitively, the order in *Mix* mixins is fixed at definition time, while MM^2 allows later reordering of input components. For instance, the *Mix* basic mixin $e_1 = \langle X \rhd x = \bullet, Y \rhd y = \bullet \rangle$ cannot be composed with $e_2 = \langle !Y \rhd y = 0, !X \rhd x = 0 \rangle$, which is unfortunate. The equivalent composition is well-typed in MM^2.

The importance of this loss of flexibility has to be further investigated. Intuitively, it can only be annoying when a mixin is reused in an unexpected way, which makes the initial order incorrect. Unfortunately, the classical examples using mixins modules (and modules in general) generally show the modular decomposition of programs, not really the reuse of existing code, with possibly badly ordered components. This comes from the lack of extensive practice of any system with mixin modules, which is one of our priorities for future work.

6 Future work

Type components and subtyping. Before to incorporate mixin modules into a practical language, we have to refine our type system in at least two respects. First, we have to design an extended version of *Mix* including ML-style user-defined type components and data types. This task should benefit from recent advances in the design of recursive module systems [6, 7]. Second, we have to enrich our type system with a notion of subtyping over mixin modules. Indeed, it might turn out too restrictive for a module system to require, as *Mix* does, a definition filling a declaration of type τ to have exactly type τ.

Compilation. The *Mix* language features anonymous definitions, and thus the compilation scheme for mixin modules proposed by Hirschowitz and Leroy [14] does not apply. A possible extension of this scheme to anonymous definitions is sketched in later work [12], but not formalized. This extension might apply to *Mix*. However, it should be possible to do better than this, by taking advantage of the more rigid structure of *Mix* mixin modules.

Other definitions of composition. The composition operator of *Mix* is somewhat arbitrary. This gives the idea to explore other definitions, perhaps more flexible. Ideally, the system should be parameterized over the notion of composition.

References

[1] D. Ancona, S. Fagorzi, E. Moggi, E. Zucca. Mixin modules and computational effects. In *Int'l Col. on Automata, Lang. and Progr.*, 2003.

[2] D. Ancona, E. Zucca. A calculus of module systems. *J. Func. Progr.*, 12(2), 2002.

[3] A. W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

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

[5] L. Cardelli. Program fragments, linking, and modularization. In *24th symp. Principles of Progr. Lang.* ACM Press, 1997.

[6] K. Crary, R. Harper, S. Puri. What is a recursive module? In *Prog. Lang. Design and Impl.* ACM Press, 1999.

[7] D. R. Dreyer, R. Harper, K. Crary. Toward a practical type theory for recursive modules. Technical Report CMU-CS-01-112, Carnegie Mellon University, Pittsburgh, PA, 2001.

[8] D. Duggan, C. Sourelis. Mixin modules. In *Int. Conf. on Functional Progr.* ACM Press, 1996.

[9] M. Flatt. PLT MzScheme: language manual. Technical Report TR97-280, Rice University, 1997.

[10] M. Flatt, M. Felleisen. Units: cool modules for HOT languages. In *Prog. Lang. Design and Impl.* ACM Press, 1998.

[11] R. Harper, M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symp. Principles of Progr. Lang.* ACM Press, 1994.

[12] T. Hirschowitz. *Modules mixins, modules et récursion étendue en appel par valeur.* PhD thesis, University of Paris VII, 2003.

[13] T. Hirschowitz. Rigid mixin modules. Research report RR2003-46, ENS Lyon, 2003.

[14] T. Hirschowitz, X. Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, ed., *Europ. Symp. on Progr.*, vol. 2305 of *LNCS*, 2002.

[15] T. Hirschowitz, X. Leroy, J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *Princ. and Practice of Decl. Prog.* ACM Press, 2003.

[16] T. Hirschowitz, X. Leroy, J. B. Wells. A reduction semantics for call-by-value mixin modules. Research report RR-4682, Inria, 2003.

[17] X. Leroy, D. Doligez, J. Garrigue, J. Vouillon. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/`, 1996–2003.

[18] E. Machkasova, F. A. Turbak. A calculus for link-time compilation. In *Europ. Symp. on Progr.*, vol. 1782 of *LNCS*. Springer-Verlag, 2000.

[19] R. Milner, M. Tofte, D. MacQueen. *The Definition of Standard ML.* The MIT Press, 1990.

[20] J. B. Wells, R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Europ. Symp. on Progr.*, vol. 1782 of *LNCS*. Springer-Verlag, 2000.