# A Reduction Semantics for Call-by-value Mixin Modules

Tom Hirschowitz
INRIA Rocquencourt
Tom.Hirschowitz@inria.fr

Xavier Leroy
INRIA Rocquencourt
Xavier.Leroy@inria.fr

J. B. Wells
Heriot-Watt University
jbw@macs.hw.ac.uk

**Abstract**

Mixin modules are a framework for modular programming that supports code parameterization, incremental programming via late binding and redefinitions, and cross-module recursion. In this paper, we develop a language of mixin modules that supports call-by-value evaluation, and formalize a reduction semantics and a sound type system for this language.

## 1 Introduction

Code structuring and reuse is an important aspect of software engineering. It can be achieved in two ways: via linguistic abstractions, such as higher-order functions, classes and objects; and via extra-linguistic mechanisms and policies such as compilation units, libraries, and linking. Module languages [13, 18, 6, 22] bridge these two approaches. On the one hand, module languages extend a largely arbitrary base programming language with linguistic constructs supporting code decomposition and reuse; these constructs can be described with mostly standard formalisms such as operational semantics and type systems. On the other hand, module languages can capture the notion of compilation unit, and express strategies for separate compilation and linking.

Three important features of a module language are (1) *parameterization*, which allows reusing a module in different contexts; (2) *overriding and late binding*, which supports incremental programming by refinements of existing modules; and (3) *cross-module recursion*, which allows definitions to be spread across several modules, even if they mutually refer to each other. Many programming languages provide two of these features, but not all three: class-based object-oriented languages provide (2) and (3), but are weak on parameterization (1); conventional linkers, as well as linking calculi [6], have cross-module recursion built in, and sometimes provide facilities for overriding, but lack parameterization; finally, ML functors and Ada generics provide powerful parameterization mechanisms, but prohibit cross-module recursion and offer no direct support for late binding.

*Mixin modules* are an alternative, hybrid framework for modularization that satisfies the three criteria above. A mixin module is a collection of named components, either defined (bound to a definition) or deferred (declared without definition). The basic operation on mixin modules is the *composition*, written with $+$, which takes two mixin modules and connects the defined components of one with the similarly-named deferred components of the other; this provides natural support for cross-module recursion. A mixin module can be bound to an identifier and composed several times with different mixin modules; this allows powerful parameterization, including but not restricted to an encoding of ML functors. The definitions inside mixin modules are late bound by default, but the *freezing* operation can make them early bound. Definitions can also be renamed and deleted. All this provides support for incremental programming.

This paper introduces a language of call-by-value mixin modules, featuring flexible control over the order of evaluation, guards on ill-founded recursion, a simple reduction semantics, and a sound type system.

**Call-by-value** In call-by-name languages, modules contain computations that are triggered by component selection. This standpoint is adopted by *CMS*, Ancona and Zucca's call-by-name proposal for mixin modules [2], and also in the **m**-calculus of Wells and Vestergaard [23]. In a call-by-value setting, it is preferable that accessing a module component does not trigger computations, and therefore that such accesses be restricted to fully evaluated modules. Thus, we need a way to trigger at once all computations of a mixin module. In [15], the proposed solution consists of evaluating mixin components as soon as possible, when they no longer depend on any deferred components. However, it results in a counter-intuitive order of evaluation. Specifically, if the base language contains imperative constructs, almost any side-effect can occur during freezing. Here, we choose to separate mixin modules, containing unevaluated code, from modules, whose contents are fully evaluated, and can then be accessed by other parts of the program. This separation is reminiscent of the separation between objects and classes in class-based languages. We introduce a close operator for triggering the evaluation of a mixin module without deferred components and transforming it into a module.

**Order of evaluation** The introduction of the close operator does not fully address the issue of controlling the evaluation order in an imperative setting: this operator fixes the time when side-effects can occur, but not the order in which they occur inside a mixin module. A simple solution is to fix the order in which close evaluates the components of mixin modules, say from left to right. However, this considerably restricts the expressive power of the language: during a composition $A+B$, components of $A$ can no longer refer to values of components of $B$. This problem is apparent in the recent proposal by Ancona et al. [1], which controls evalua-

tion order in *CMS* via the addition of monadic bindings. To preserve the flexibility of the original mixin composition operator, yet allow the programmer to control the evaluation order when needed, the approach we follow in this paper is to allow the close operator to reorder the components of a mixin according to their dependencies, and provide the programmer with the ability to indicate additional "fake" dependencies between mixin components, in order to constrain the evaluation order.

**Recursion** Many programming languages restrict recursive definitions, either to make them easier to compile, or to statically reject ill-founded recursive definitions. For example, Standard ML restricts recursive definitions to syntactic functions, while C and OCaml also allow data constructor applications in the right-hand sides of recursive definitions. With mixin modules, arbitrary recursive definitions can appear dynamically when a composition connects a deferred component of one of its arguments to a defined component of the other argument. There are two ways to address this issue: either add support for these arbitrary recursive definitions in the base language, e.g., through the use of call-by-name or lazy evaluation for intra-mixin references [2]; or statically rule out the mixin compositions that could result in recursive definitions that are not supported by the base language [15]. We believe that adding mixin modules to a base language should not significantly modify its dynamic semantics and compilation scheme, and thus prefer the latter approach: by enriching the type system with dependency information between mixin components, we ensure that mixin composition never results in recursive definitions that cannot be compiled as in the base language.

**Reduction semantics and type soundness** The semantics of our language is given in terms of a source-level reduction semantics. This improves over our earlier work on $CMS_v$ [15], whose semantics were defined by type-directed translation to a $\lambda$-calculus with let rec. In particular, the reduction semantics makes it easier to prove the soundness of the type system, using the standard subject reduction and progress properties.

The remainder of this paper is organized as follows. An overview of the mixin language is presented in section 2. Section 3 defines the calculus, named *MM*, and its semantics. A type system is formalized in section 4, and it is proved sound. We review related work in section 5, and directions for future work in section 6. Proofs and inessential details of the formalization are omitted from this paper, but can be found in the companion technical report [17].

## 2 Overview of the mixin language

### 2.1 Mixin modules

A mixin module is an unordered, unevaluated, possibly incomplete module: it is a set of named definitions and declarations. Consider the following mixin module, written in an ML-style syntax:

```
mixin A =
  import
    val x : int
    val f : int -> int
  export
    define y = g(0) + x
    define g z = ... f ...
  end
```

The declaration `val x : int` is used by the definition `define y = g(0) + x`. The declaration `val f : int -> int` is used by the definition `define g z = ... f ...` The scope of the definitions is the entire mixin module, as illustrated by the definition `define y = g(0) + x`, depending on `g`.

The operator for linking mixin modules is composition `+`, which combines two mixin modules, filling the declarations of one argument with the definitions of the other, and conversely. Consider the following mixin module.

```
mixin B =
  import
    val y : int
    val g : int -> int
  export
    define x = y + 1
    define f z = ... g ...
  end
```

The composition `mixin C = A + B` of `A` and `B` is equivalent to the mixin module:

```
mixin C =
  import
  export
    define y = g(0) + x
    define g z = ... f ...
    define f z = ... g ...
    define x = y + 1
  end
```

The declarations of one mixin module are replaced with the similarly named definitions of the other. The export section is the concatenation of the export sections of `A` and `B`. The code remains unevaluated, so the evaluation of `C` does not go wrong. However, there is an ill-founded recursion between `x` and `y`, and if we try to evaluate the definitions contained by `C`, we will obtain either non-termination (in a call-by-name setting) or a run-time error (in a call-by-value setting).

Mixin modules feature late binding: definitions can be overriden by deletion followed by composition. The delete operator `|-` removes one definition from a mixin. For instance, `mixin B' = B |- x` is equivalent to

```
mixin B' =
  import
    val x : int
    val y : int
    val g : int -> int
  export
    define f z = ... g ...
  end
```

A new definition for `x` can be provided in another mixin module.

```
mixin D = import
          export
            define x = 0
          end
```

The mixin module `E = A + B' + D` is equivalent to

```
mixin E = import
          export
            define y = g(0) + x
            define g z = ... f ...
            define f z = ... g ...
            define x = 0
          end
```

Mixin `E` has no remaining imports, thus it can be instantiated to a normal module using the `close` operator. The semantics of `close` includes a reordering of the definitions of the mixin being closed, in order to avoid references to a not yet evaluated definition. The initial ordering is kept whenever possible. In the case of `E`, reordering moves the definition of `y` to the end, because it needs the values of `g` and `x` (and possibly `f`) to evaluate. The definition `module M = close E` is thus equivalent to:

```
module M = struct
  let rec g z = ... f ...
          f z = ... g ...
  let x = 0
  let y = g(0) + x
end
```

The evaluation of `M` consists in successively evaluating the definitions, and returning the evaluated module:

```
module M = struct
  let rec g z = ... f ...
          f z =  ... g ...
  let x = 0
  let y = v
end
```

where $v$ is the value of the expression `g(0) + x`.

We refer to [5, 23, 3, 15] for more details on mixin modules and mixin operators.

### 2.2 An extended binding construct

Consider again the translation from mixin `E` to module `M` outlined above. It would have been simpler to write `M` as follows:

```
module M = struct
  let rec g z = ... f ...
          f z = ... g ...
          x = 0
          y = g(0) + x
end
```

However, ML does not allow the definitions of `x` and `y` to be included in the same mutually recursive definition as `f` and `g`. Indeed, the `let rec` construct of ML only allows to bind syntactic functions (or constructed values in the case of OCaml). Therefore, the effect of closing a mixin module cannot in general be expressed by a single ML-style `let rec`; a sequence of `let` and `let rec` bindings is required. As shown in [15], such sequences can be derived systematically from the results of a dependency analysis. However, this construction is involved and hard to reason about.

To avoid these complications, our calculus features a slightly more powerful `let rec` than that of ML, reminiscent of monadic recursive bindings [9]. It evaluates the definitions from left to right, and only goes wrong when the value of a variable defined to the right of the current definition is needed. For instance, the `let rec` over `g`, `f`, `x` and `y` shown above evaluates correctly, because `g`, `f`, and `x` are bound to syntactic values, and the expression defining `y` is evaluated last.

Notice that the body of `g` makes a reference to `f`, which is defined to the right of `g`. We call such a reference a *forward reference*. A forward reference is syntactically correct if it refers to an expression of *predictable shape*. This notion of predictable shape will be formally defined in section 3; for now on, it suffices to know that a function abstraction is an expression of predictable shape. Thus, in the example above, the forward reference from `g` to `f` is correct because `f` is bound to a function abstraction.

Operationally, a forward reference to a variable `f` in the definition of `g` can yield an error. For example, if `g` is defined as `f(0)` (instead of being defined as a function as in the example), it is not permitted to use the (possibly not yet evaluated) definition of `f`, so the evaluation gets stuck. Conversely, if during the evaluation of `g`, `f` appears only in non-strict contexts, then it does not yield any error. An instance of this nice case is when `g` is already a value.

### 2.3 Typing issues

Our extended `let rec` is not much more powerful than that of ML. Its main interest is that complex series of interleaved `let` bindings and mutually recursive `let rec` bindings are now written as straightforward definitions. Its typing is much less straightforward of course, since it requires the analysis of dependencies between the definitions. This analysis has to go beyond immediate dependencies, as shown by the following example.

**Example 1** *Consider the following binding, where braces enclose records and* `X` *and* `Z` *are record field names.*

```
let rec x = { X = z }
        y = x.X.Z
        z = { Z = 0 }
```

*There is a forward reference from* `x` *to* `z`*, but the definition of* `z` *is of predictable shape, so the expression is syntactically correct. Moreover, there are no forward references needing the value of the referenced definition. One could expect it to be a sufficient condition for the binding not to go wrong because of dependencies. Unfortunately, the evaluation of the definition of* `y` *needs both the values of* `x` *and* `z`*.*

Roughly, the correct requirement is that no forward reference path starts with a strict dependency. We say that a definition `x = M` strictly depends on another one `y = N`, when the evaluation of `M` might require the value of `y`. What does "might require" mean here? It is a very restrictive syntactic approximation: the only case where we detect that an expression `M` will not need the value of one of its free variables `x` is when `M` is a value of predictable shape. In example 1, there is a forward reference path from `x` to `z`, which does not end with a strict dependency, since `{ X = z }` is a value of predictable shape. However, this path extends to a forward reference path from `y` to `z`, which starts with a strict dependency. Therefore, the binding is rejected by the type system. A non-strict dependency is called a safe dependency.

In our system, `let rec` bindings appear as the result of evaluating `close` operations. To ensure statically that such bindings are correct, the type system keeps track of the dependencies between mixin components. The type of a

| | | | |
|---|---|---|---|
| $x$ | $\in$ | *Vars* | Variable |
| $X$ | $\in$ | *Names* | Name |
| Expressions: | | | |
| $e$ | ::= | $x$ | Variable |
| | \| | $\{X_1 = e_1 \dots X_n = e_n\}$ | Record |
| | \| | $e.X$ | Select |
| | \| | $\mathsf{let\ rec}\, x_1 = e_1 \dots x_n = e_n$ $\mathsf{in}\, e$ | let rec |
| | \| | $\langle X_1 \triangleright x_1 \dots X_n \triangleright x_n;$ $d_1 \dots d_m \rangle$ | Structure |
| | \| | $e_1 + e_2$ | Compose |
| | \| | $\mathsf{close}\, e \mid e\,!\,X$ | Close, freeze |
| | \| | $e_{\mid X_1 \dots X_n} \mid e_{\mid -X_1 \dots X_n}$ | Project, delete |
| | \| | $e_{:X_1 \dots X_n} \mid e_{:-X_1 \dots X_n}$ | Show, hide |
| | \| | $e[X_1 \mapsto Y_1 \dots X_n \mapsto Y_n]$ | Rename |
| | \| | $e_{X \succ Y}$ | Split |
| Definitions: | | | |
| $d$ | ::= | $X[x_1 \dots x_n] \triangleright x = e$ | Named |
| | \| | $\_[x_1 \dots x_n] \triangleright x = e$ | Anonymous |

Figure 1: Syntax of *MM*

| | | | |
|---|---|---|---|
| $s$ | ::= | $X_1 = e_1 \dots X_n = e_n$ | Record |
| $b$ | ::= | $x_1 = e_1 \dots x_n = e_n$ | Binding |
| $\iota$ | ::= | $X_1 \triangleright x_1 \dots X_n \triangleright x_n$ | Input |
| $o$ | ::= | $d_1 \dots d_2$ | Output |
| $r$ | ::= | $X_1 \mapsto Y_1 \dots X_n \mapsto Y_n$ | Renaming |
| $op[e]$ | ::= | $e.X$ | Operator |
| | \| | $\mathsf{close}\, e \mid e\,!\,X$ | |
| | \| | $e_{\mid X_1 \dots X_n} \mid e_{\mid -X_1 \dots X_n}$ | |
| | \| | $e_{:X_1 \dots X_n} \mid e_{:-X_1 \dots X_n}$ | |
| | \| | $e[X_1 \mapsto Y_1 \dots X_n \mapsto Y_n]$ | |
| | \| | $e_{X \succ Y}$ | |

For a finite map $f$, and a set of variables $P$,
$dom(f)$ is its domain,
$cod(f)$ is its codomain
$f_{\mid P}$ is its restriction to $P$,
$f_{\setminus P}$ is its restriction to $Vars \setminus P$.

Figure 2: Meta-variables and notations

mixin contains both type information about its components, and a graph representing their dependencies. For example, the type of the mixin module `A` above is:

```
mixin A :
  import
    val x : int
    val f : int -> int
  export
    val y : int { g:0, x:0 }
    val g : int -> int { f:1 }
  end
```

where `{ g:0, x:0 }` indicates that `y` strictly depends on `g` and `x`, and `{ f:1 }` indicates that `g` safely depends on `f`.

When composing two mixin modules, the type system takes the union of their dependency graphs. When a concrete mixin (a mixin with no declarations, only definitions) is instantiated, its graph is required not to have cycles with strict dependencies. This is sufficient: if there is no cycle with strict dependencies, then an ordering of definitions can be found, such that no forward reference path starts with a strict dependency. The `close` operator finds this ordering, and reduces to the corresponding `let rec` binding.

## 3 Definition of the language

### 3.1 Syntax

The syntax of *MM* is defined in figure 1. The meta-variables $X$ and $x$ range over names and variables, respectively. Variables are used as binders, as usual. Names are used to refer to the definitions contained in a mixin module; they provide the external interface to this mixin module. Figure 2 recapitulates the meta-variables and notations we introduce in the remainder of this section.

Expressions include variables $x$, records (labeled by names) $\{X_1 = e_1 \dots X_n = e_n\}$, and record selection $e.X$, which are standard.

*MM* features mutually recursive bindings of the shape $\mathsf{let\ rec}\ b\ \mathsf{in}\ e$ (where $b$ is a list of definitions $x_1 = e_1 \dots x_n = e_n$). There are no syntactic restrictions on the right-hand sides of bindings; in particular, they are not required to be syntactic values.

Expressions also include mixin structures. A structure is a pair of an input $\iota$ of the shape $X_1 \triangleright x_1 \dots X_n \triangleright x_n$, and of an output $o$ of the shape $d_1 \dots d_m$. $\iota$ maps external names imported by the structure to internal variables (used in $o$). $o$ is an ordered list of definitions $d$. A definition is of the shape $L[x_1 \dots x_n] \triangleright x = e$, where the label $L$ may be either a name $X$ or the anonymous label $\_$ and $e$ is the body of the definition. The possibly empty finite set of names $x_1 \dots x_n$ is the set of user-requested dependencies of this definition on other definitions of the structure. These additional dependencies allow the programmer to force an order of evaluation.

Finally, *MM* features a set of operators over mixins similar to those described in [5, 3, 15]: composition $e_1 + e_2$, closure $\mathsf{close}\, e$, freezing $e\,!\,X$, projection $e_{\mid X_1 \dots X_n}$, deletion $e_{\mid -X_1 \dots X_n}$, showing $e_{:X_1 \dots X_n}$, hiding $e_{:-X_1 \dots X_n}$, and renaming $e[X_1 \mapsto Y_1 \dots X_n \mapsto Y_n]$. There is a new operator called splitting $e_{X \succ Y}$, which we will describe in section 3.2. We let $op$ range over the set of operators (see figure 2), and denote by $op[e]$ the application of $op$ to the expression $e$.

**Syntactic constraints** Renamings $r = (X_1 \mapsto Y_1 \dots X_n \mapsto Y_n)$, inputs $\iota = (X_1 \triangleright x_1 \dots X_n \triangleright x_n)$, records $s = (X_1 = e_1 \dots X_n = e_n)$, and bindings $b = (x_1 = e_1 \dots x_n = e_n)$ are required to be finite maps: a renaming is a finite map from names to names, an input is a finite map from names to variables, a record is a finite map from names to expressions, and a binding is a finite map from variables to expressions. Requiring them to be finite maps means that they should not bind the same variable or name twice. Renamings and inputs are required to be injective. Outputs $o = (d_1 \dots d_n)$ are required not to define the same name twice, and not to define the same variable twice. Structures are required not to define the same name twice and not to define the same variable twice. User-requested dependencies in a definition must be bound in the same structure.

In a $\mathsf{let\ rec}$ binding $b = (x_1 = e_1 \dots x_n = e_n)$, we say that there is a forward reference from $x_i$ to $x_j$ if $1 \leq i \leq j \leq n$ and $x_j \in FV(e_i)$, where $FV(e_i)$ denotes the free variables of $e_i$. A forward reference from $x_i$ to $x_j$ is syntactically forbidden, except when $e_j$ is of predictable shape. An expression of predictable shape is a structure, a record, or a binding followed by an expression of predictable shape. Formally

$$\frac{dom(b) \perp FV(\mathbb{L})}{\mathbb{L}[\mathsf{let\ rec}\ b\ \mathsf{in}\ e] \rightsquigarrow_c \mathsf{let\ rec}\ b\ \mathsf{in}\ \mathbb{L}[e]} \quad (\textsc{Lift}) \qquad \{X_1 = v_1 \dots X_n = v_n\}.X_i \rightsquigarrow_c v_i \quad (\textsc{Select})$$

$$\langle \iota; o\rangle_{|-X_1 \dots X_n} \rightsquigarrow_c \langle \iota, Input(o)_{|\{X_1 \dots X_n\}}; o_{\setminus\{X_1 \dots X_n\}}\rangle \quad (\textsc{Delete})$$

$$\langle \iota; o\rangle_{|X_1 \dots X_n} \rightsquigarrow_c \langle \iota, Input(o)_{\setminus\{X_1 \dots X_n\}}; o_{|\{\_, X_1 \dots X_n\}}\rangle \quad (\textsc{Project}) \qquad \langle \iota; o\rangle_{:X_1 \dots X_n} \rightsquigarrow_c \langle \iota; Show(o, \{X_1 \dots X_n\})\rangle \quad (\textsc{Show})$$

$$\langle \iota; o\rangle_{:-X_1 \dots X_n} \rightsquigarrow_c \langle \iota; Show(o, Names \setminus \{X_1 \dots X_n\})\rangle \quad (\textsc{Hide})$$

$$\langle \iota; o_1, X[y^*] \triangleright x = e, o_2\rangle \,!\, X \rightsquigarrow_c \langle \iota; o_1, \_[y^*] \triangleright x = e, o_2, X \triangleright \_ = x\rangle \quad (\textsc{Freeze})$$

$$\frac{Names(\langle \iota; o\rangle) \perp (cod(r) \setminus dom(r))}{\langle \iota; o\rangle[r] \rightsquigarrow_c \langle \iota\{r\}; o\{r\}\rangle} \quad (\textsc{Rename}) \qquad \langle \iota; o_1, X[z^*] \triangleright x = e, o_2\rangle_{X \succ Y} \rightsquigarrow_c \langle \iota, X \triangleright x; o_1, Y[z^*] \triangleright \_ = e, o_2\rangle \quad (\textsc{Split})$$

$$\frac{\langle \iota_1; o_1\rangle \Subset \langle \iota_2; o_2\rangle \qquad Names(o_1) \perp Names(o_2)}{\langle \iota_1; o_1\rangle + \langle \iota_2; o_2\rangle \rightsquigarrow_c \langle (\iota_1 \cup \iota_2) \setminus Input(o_1, o_2); o_1, o_2\rangle} \quad (\textsc{Sum}) \qquad \frac{Bind(\overline{o}) \text{ is correct}}{\mathsf{close}\langle \emptyset; o\rangle \rightsquigarrow_c \mathsf{let\ rec}\, Bind(\overline{o})\, \mathsf{in}\, Record(\overline{o})} \quad (\textsc{Close})$$

Figure 3: Contraction relation

$e_\downarrow \in Predictable ::= \{o\} \mid \langle \iota; o\rangle \mid \mathsf{let\ rec}\ b\ \mathsf{in}\ e_\downarrow$.

**Sequences** Outputs may be viewed as finite maps from pairs of a label and a variable $(L, x)$ to pairs of a finite set of variables $(x_1 \dots x_n)$ and an expression $e$. Renamings, inputs, records, bindings, and outputs are often considered as finite maps in the sequel. We refer to them collectively as sequences, and use the usual notations on finite maps, such as the domain *dom*, the codomain *cod*, the restriction $\cdot_{|P}$ to a set $P$, or the co-restriction $\cdot_{\setminus P}$ outside of a set $P$. Notice that the domain of an output $o$, restricted to pairs of a name and a variable (no anonymous label), may in turn be viewed as an input, since it is an injective finite map. We denote it by $Input(o)$.

**Structural equivalence** We consider the expressions equivalent up to alpha-conversion of variables bound in structures and let rec expressions. In the following, we assume that no variable capture occurs.

### 3.2 Semantics

The semantics of *MM* is defined in two steps: a contraction relation describes the action of the operators, and a reduction relation extends it to any expression.

As defined in figure 4, an *MM* value is either a variable $x$, an evaluated record $\{X_1 = v_1 \dots X_1 = v_1\}$, or a structure $\langle \iota; o\rangle$. A valid result of the evaluation of an *MM* expression is a value, possibly surrounded by an evaluated binding. It thus has the shape $\mathsf{let\ rec}\ x_1 = v_1 \dots x_n = v_n\ \mathsf{in}\ v$. The meta-variables $s_v$ and $b_v$ respectively range over evaluated record sequences and bindings.

#### 3.2.1 The contraction relation

The contraction relation $\rightsquigarrow_c$ is defined by the rules in figure 3, where for any sets $P_1 \dots P_n$, $P_1 \perp \dots \perp P_n$ means that the $P_i$'s are pairwise disjoint. It uses notions defined in figures 4 and 5.

| | | | |
|---|---|---|---|
| Value: | $v$ | $::=$ | $x \mid \{s_v\} \mid \langle \iota; o\rangle$ |
| Answer: | $a$ | $::=$ | $v \mid \mathsf{let\ rec}\ b_v\ \mathsf{in}\ v$ |
| Value sequence: | $s_v$ | $::=$ | $X_1 = v_1 \dots X_1 = v_1$ |
| | $b_v$ | $::=$ | $x_1 = v_1 \dots x_n = v_n$ |

Figure 4: Values in *MM*

The first rule Lift describes how let rec bindings are lifted up to the top of the term. When the evaluation of a sub-expression results in a let rec binding, *MM* lifts it one level up, as follows. Lift contexts $\mathbb{L}$ are defined as

$$\begin{array}{lcl} \mathbb{L} & ::= & \{\mathbb{S}\} \mid op[\Box] \mid \Box + e \mid v + \Box \\ \mathbb{S} & ::= & s_v, X = \Box, s. \end{array}$$

Rule Lift states that an expression of the shape $\mathbb{L}[\mathsf{let\ rec}\ b\ \mathsf{in}\ e]$ evaluates to $\mathsf{let\ rec}\ b\ \mathsf{in}\ \mathbb{L}[e]$, provided no variable capture occurs.

The record selection rule Select straightforwardly describes the selection of a record field.

The rules for mixin deletion Delete and projection Project are dual. Rule Delete describes how *MM* deletes a finite set of names $X_1 \dots X_n$ from a structure $\langle \iota; o\rangle$. First, $o$ is restricted to the other definitions, to obtain $o_{\setminus\{X_1 \dots X_n\}}$ (which is shorthand for $o_{\setminus\{X_1 \dots X_n\} \times Vars}$). Second, the removed definitions remain bound as inputs, by adding the corresponding inputs to $\iota$.

Rule Project describes how *MM* projects a mixin to some finite set of names $X_1 \dots X_n$ from a structure $\langle \iota; o\rangle$. First, $o$ is restricted to the corresponding definitions and to the local ones, to obtain $o_{|\{\_, X_1 \dots X_n\}}$ (which is a shorthand for $o_{|\{\_, X_1 \dots X_n\} \times Vars}$). Then, the removed definitions remain bound as inputs, by adding the corresponding inputs to $\iota$.

Rules Show and Hide are dual. Rule Show allows to hide all the exported names of a structure, except the given ones. It proceeds by making the other definitions local, as defined by

$$Show(L[y^*] \triangleright x = e, \mathcal{N}) = \begin{cases} L[y^*] \triangleright x = e \text{ if } L \in \mathcal{N} \\ \_[y^*] \triangleright x = e \text{ otherwise.} \end{cases}$$

$\langle \iota_1; o_1 \rangle \fatsemi \langle \iota_2; o_2 \rangle$ means $\begin{cases} \langle \iota_1; o_1 \rangle \doteq \langle \iota_2; o_2 \rangle \text{ and} \\ \langle \iota_2; o_2 \rangle \doteq \langle \iota_1; o_1 \rangle. \end{cases}$

$\langle \iota_1; o_1 \rangle \doteq \langle \iota_2; o_2 \rangle$ means that
for all $(L \triangleright x) \in dom(\langle \iota_1; o_1 \rangle)$,
$x \in FV(o_2) \cup Variables(\langle \iota_2; o_2 \rangle)$
$\Rightarrow (L \triangleright x) \in dom(\langle \iota_2; o_2 \rangle)$ and $L \in Names$.

Figure 5: Definition of $\fatsemi$

Rule HIDE symmetrically hides the given names in a structure. It proceeds by showing the other ones.

Rule FREEZE describes how a name $X$ is frozen in a structure $\langle \iota; o \rangle$. First, the corresponding definition $X[y^*] \triangleright x = e$ is made local, by replacing $X$ with the local label $\_$. Then, a new definition is added at the end of the output. It is named $X$, is bound to a fresh variable (denoted by $\_$ in the rule by abuse of notation), and is defined as equal to $x$.

Renaming of a structure $\langle \iota; o \rangle$ by a renaming $r$, defined by rule RENAME, replaces the names in $\iota$ and $o$ with the new ones. Formally, for $\mathcal{N} \subset Names$, we define $r_{\mathcal{N}}$ by $r \cup id_{|\mathcal{N} \setminus dom(r)}$ and for a finite map $f$ with $dom(f) \subset Names$, we define $f\{r\}$ by $f \circ (r_{dom(f)})^{-1}$. The finite map $f\{r\}$ is well-defined provided $r_{dom(f)}$ is injective, which holds as soon as $cod(r) \cap dom(f) \subset dom(r)$ or in other words $dom(f) \perp (cod(r) \setminus dom(r))$. By the side-condition $Names(\langle \iota; o \rangle) \perp (cod(r) \setminus dom(r))$, this is the case for $\iota\{r\}$. (We denote by $Names(\langle \iota; o \rangle)$ the set of names bound by the structure, i.e. $dom(\iota) \cup dom(Input(o))$.) Finally, we define $o\{r\}$ by $o \circ (r_{Names(o)}, id_{Vars})^{-1}$, ordered like $o$, and where $(f_1, f_2)(x_1, x_2) = (f_1(x_1), f_2(x_2))$. Notice that when composing two functions $f \circ g$, we consider a function whose domain is $g^{-1}(dom(f))$ and on this domain is $f(g(x))$. In the rule, $o\{r\}$ is well defined, thanks to the side condition. Syntactic correctness is preserved, since $r_{Names(\langle \iota; o \rangle)}$ is injective. Hence, after renaming, no name is defined twice.

The SPLIT rule introduces a new operator "split". If there is a definition $X[z^*] \triangleright x = e$ for the name $X$ in $\langle \iota; o \rangle$, the split operator $\langle \iota; o \rangle_{X \succ Y}$ splits it into an input $X \triangleright x$ and a definition $Y[z^*] \triangleright y = e$ (with a fresh $y$). References to $x$ continue referencing it as an input, but the former definition $e$ remains exported as $Y$. The operation is different from renaming $X$ to $Y$ or deleting $X$.

The SUM rule defines the composition of two structures $\langle \iota_1; o_1 \rangle$ and $\langle \iota_2; o_2 \rangle$. The result is a structure $\langle \iota; o \rangle$, defined as follows. $\iota$ is the union of $\iota_1$ and $\iota_2$, where names defined in $o_1$ or $o_2$ are removed. $o$ is defined as the concatenation of $o_1$ and $o_2$. The side condition $\langle \iota_1; o_1 \rangle \fatsemi \langle \iota_2; o_2 \rangle$ checks that both structures agree on bound variables, and that no free variable is captured. It is defined in figure 5, where $dom(\langle \iota; o \rangle) = \iota \cup dom(o)$, and $Variables(\langle \iota; o \rangle)$ denotes $cod(\iota) \cup \{x \mid (L, x) \in dom(o)\}$. Lastly, $o_1$ and $o_2$ are required not to define the same names.

Finally, the CLOSE rule describes the instantiation of a structure $\langle \iota; o \rangle$. $\iota$ must be empty. The instantiation is in three steps. First, $o$ is reordered into $\overline{o}$, according to a combination of its dependencies, its user-requested dependencies, and its default ordering. This reordering operation is defined below in section 3.2.3. Second, a binding $Bind(\overline{o})$ is generated, defining, for each definition $L[y^*] \triangleright x = e$ in $o$, the binding $x = e$. The order of definitions in $\overline{o}$ is preserved. Third, the named definitions of $\overline{o}$ are put in a record $Record(\overline{o})$ containing a field $X = x$ for each named definition $X[y^*] \triangleright x = e$ in $\overline{o}$. As a side condition, the CLOSE rule requires that the generated binding $Bind(\overline{o})$ is syntactically correct, especially that it contains no forward reference to bindings of unpredictable shapes.

Evaluation context:
$\mathbb{E} ::= \mathbb{F}$
$\mid$ let rec $b_v$ in $\mathbb{F}$
$\mid$ let rec $\mathbb{B}[\mathbb{F}]$ in $e$

Lift context:
$\mathbb{L} ::= \{\mathbb{S}\} \mid op[\Box] \mid \Box + e \mid v + \Box$

Nested lift context:
$\mathbb{F} ::= \Box \mid \mathbb{L}[\mathbb{F}]$

Binding context:
$\mathbb{B} ::= b_v, x = \Box, b$

Record context:
$\mathbb{S} ::= s_v, X = \Box, s$

Figure 6: Evaluation contexts

$$(\text{let rec } b_v \text{ in } \mathbb{F})(x) = b_v(x) \quad \text{(EA)}$$
$$(\text{let rec } b_v, y = \mathbb{F}, b \text{ in } e)(x) = b_v(x) \quad \text{(IA)}$$

Figure 7: Access in evaluation contexts

### 3.2.2 The reduction relation

The reduction relation is defined by the rules in figure 8, using notions defined in figures 6 and 7.

Rule CONTEXT extends the contraction relation to any evaluation context. Evaluation contexts are defined in figure 6. We call a nested lift context $\mathbb{F}$ a series of nested lift contexts. An evaluation context $\mathbb{E}$ is a nested lift context, possibly inside a partially evaluated binding, or under a fully evaluated binding. This unusual formulation of evaluation contexts is intended to enforce determinism of the reduction relation. The idea is that evaluation never takes place inside or under a let rec, except the topmost one. Other bindings inside the expression have first to be lifted to the top by rule LIFT, and then merged with the topmost let rec if any, by rules EM and IM. In the case where the topmost binding is of the shape $b_v, x = (\text{let rec } b_1 \text{ in } e), b_2$, rule IM allows to merge $b_1$ with the current binding. When an inner binding has been lifted to the top, if there is already a topmost binding, then the two bindings are merged together by rule EM. As a result, when the evaluation encounters a binding, it is always possible to lift it up to the top and then merge it with the topmost binding if any.

Finally, rule SUBST describes the use of bound values when needed. The notion of a needed value is formalized by strict contexts, which are defined by

$$\mathbb{N} ::= op[\Box] \mid \Box + v_1 \mid v_2 + \Box \quad (v_2 \text{ is not a variable}).$$

In *MM* the value of a variable is copied only when needed for the application of an operator, or for composition. The value of a variable $x$ is found in the current evaluation context, by looking for the nearest binding of $x$ enclosing the reference to $x$, as formalized by the notion of access in evaluation contexts in figure 7. There are two kinds of accesses.

- In the case of a context of the shape let rec $b_v$ in $\mathbb{F}$, if the referenced variable $x$ is bound in the topmost binding $b_v$, then $b_v(x)$ is the requested value, provided the following two capture conditions hold. First, no

$$\frac{e \rightsquigarrow_c e'}{\mathbb{E}\,[e] \longrightarrow \mathbb{E}\,[e']} \quad \text{(Context)} \qquad \frac{\mathbb{E}\,[\mathbb{N}\,](x) = v}{\mathbb{E}\,[\mathbb{N}\,[x]] \longrightarrow \mathbb{E}\,[\mathbb{N}\,[v]]} \quad \text{(Subst)}$$

$$\frac{dom(b_1) \perp \{x\} \cup dom(b_v, b_2) \cup FV(b_v, b_2) \cup FV(f)}{\mathsf{let\ rec}\ \ b_v, x = (\mathsf{let\ rec}\ \ b_1\ \ \mathsf{in}\ \ e), b_2\ \ \mathsf{in}\ \ f \longrightarrow \mathsf{let\ rec}\ \ b_v, b_1, x = e, b_2\ \ \mathsf{in}\ \ f} \quad \text{(IM)}$$

$$\frac{dom(b) \perp (dom(b_v) \cup FV(b_v))}{\mathsf{let\ rec}\ \ b_v\ \ \mathsf{in}\ \ \mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ e \longrightarrow \mathsf{let\ rec}\ \ b_v, b\ \ \mathsf{in}\ \ e} \quad \text{(EM)}$$

Figure 8: Reduction relation

variable free in $b_v(x)$ should be captured by $\mathbb{F}$. Second, $x$ should not be captured by $\mathbb{F}$ either, because this would mean that another binding is concerned, inside $\mathbb{F}$.

- In the case of a context of the shape $\mathbb{E}\,[\mathsf{let\ rec}\ \ b_v, y = \mathbb{F}, b\ \ \mathsf{in}\ \ e]$, if the referenced variable $x$ is bound in the binding $b_v$, then $b_v(x)$ is the requested value, provided the following two capture conditions hold. First, no variable free in $b_v(x)$ should be captured by $\mathbb{F}$. Second, $x$ should not be captured by $\mathbb{F}$ either, because this would mean that another binding is involved, inside $\mathbb{F}$.

### 3.2.3 Reordering during instantiation

The Close rule makes use of a reordering operation on outputs $\overline{o}$, which we now define. This operation takes four aspects of its argument into account: its internal dependencies, its user-requested dependencies, the shapes of its definitions, and its original ordering. Internal dependencies and user-requested dependencies are considered imperative requirements on the final ordering: if a definition $d$ could call another definition $d'$, then $d'$ must be put before $d$ in the final ordering. The shapes of the definitions are examined in order to never generate a binding with forward references to definitions of unpredictable shape. The original ordering is only used as a hint, in the case where no constraint forces one definition to be put before the other.

**Remark 1** *The criterion on bindings mentioned in section 2, forbidding forward dependency paths starting with a strict edge, will look reversed here. Indeed, when a definition $d_1$ calls another definition $d_2$, it is also possible to see it as a constraint on their ordering, such as "the definition $d_2$ must be put before the definition $d_1$". As we will use this relation on definitions as an ordering for generating a binding, the second way is more intuitive. A consequence is that the correctness criterion now forbids backward dependency paths ending with a strict edge.*

More formally, the dependency graph of an output is defined in figure 9. For each pair of definitions $L[y^*] \triangleright x = e$ and $L'[z^*] \triangleright x' = e'$ in $o$, there can be two kinds of edges.

- If $x'$ is free in $e$, then an edge is added from $x'$ to $x$. This edge is labeled with a degree $\chi \in \{\text{☺}, \text{☹}\}$. $\chi$ is determined by $Degree(x', e)$, where the *Degree* function

$$\frac{(L[y^*] \triangleright x = e) \in o \qquad \frac{(L'[z^*] \triangleright x' = e') \in o \qquad \chi = Degree(x', e)}{x' \xrightarrow{\chi}_o x}}{}$$

$$\frac{(L[x_1 \ldots x_n] \triangleright x = e) \in o \qquad (L'[z^*] \triangleright x_i = e') \in o}{x_i \xrightarrow{\text{☹}}_o x}$$

Figure 9: Dependencies in an output

$$\frac{x \xrightarrow{\chi_1}{}^{+} z \qquad z \xrightarrow{\chi_2} y}{x \xrightarrow{\chi_2}{}^{+} y} \qquad \frac{x \xrightarrow{\chi} y}{x \xrightarrow{\chi}{}^{+} y}$$

Figure 10: Transitive closure of $\rightarrow$

is defined for $x' \in FV(e)$ by

$$\begin{array}{rcl} Degree(x', \langle \iota; o \rangle) & = & \text{☺} \\ Degree(x', \{s_v\}) & = & \text{☺} \\ Degree(x', e) & = & \text{☹ otherwise.} \end{array}$$

More sophisticated definitions of *Degree* can be given, along the lines of [4, 15]. Section 4.1 gives a minimal axiomatization of correct *Degree* functions.

- If $x'$ is mentioned in $y^*$, then an edge from $x'$ to $x$ is added, with degree ☹. Fake dependencies act as real strict dependencies.

The transitive closure of this relation is defined in figure 10, by taking the degree of a path as the degree of its last edge. The relation $\xrightarrow{\text{☹}}{}^{+}_{o}$ gives a conservative approximation of which definition needs the value of which other one in $Bind(\overline{o})$. Reordering $o$ according to $\xrightarrow{\text{☹}}{}^{+}_{o}$ is not enough, however, because the generated binding might be syntactically incorrect. Indeed, it is forbidden to make forward references to definitions of unpredictable shape inside a binding. Strict forward references to definitions of unpredictable shape already correspond to edges labeled ☹ in $\rightarrow_o$, and are therefore taken into account when reordering according to $\xrightarrow{\text{☹}}{}^{+}_{o}$. Weak forward references to definitions of unpredictable shape correspond to edges labeled ☺ in $\rightarrow_o$, and are therefore not taken into account when reordering according to $\xrightarrow{\text{☹}}{}^{+}_{o}$.

$$
\begin{array}{rcl}
M \in \mathit{Types} & ::= & \{O\} \mid \langle I; O; G\rangle \\
I, O & \in & \mathit{Names} \xrightarrow{\text{Fin}} \mathit{Types} \\
G & \subset_{\text{Fin}} & \{X \xrightarrow{\chi} Y \mid \ X, Y \in \mathit{Names}, \\
 & & \qquad \chi \in \mathit{Degrees}\} \\
\Gamma & \in & \mathit{Vars} \xrightarrow{\text{Fin}} \mathit{Types}
\end{array}
$$

Figure 12: Types

Let $\succ_o$ be the relation defined by $x_1 \succ_o x_2$ if and only if $x_1 \xrightarrow{\text{☺}} x_2$ and $o(x_1) \notin \mathit{Predictable}$. This relation captures the ordering constraints induced by weak references to definitions of unpredictable shape.

We define the binary relation $\succcurlyeq_o$ by the lexical ordering $\succcurlyeq_o = \big((\xrightarrow{\text{☺}}_o^+ \cup \succ_o)^+, >_o\big)$, where $>_o$ is the initial ordering in $o$. If $\succcurlyeq_o$ contains no cycle, $o$ is said to be correct. This is written $\vdash o$. In this case, $\overline{o}$ denotes $o$ reordered according to $\succcurlyeq_o$.

## 4 Typing

### 4.1 Type system

In this section, we present a type system for *MM*. Types are defined in figure 12. There are only two kinds of types, record types $\{O\}$ and mixin types $\langle I; O; G\rangle$, where $I$ and $O$ range over finite maps from names to types and $G$ is a finite graph over names, labeled by degrees. Such a graph is called an abstract dependency graph. (Remember that dependency graphs over the whole set of nodes are called concrete.) An environment $\Gamma$ is a finite map from variables to types. We write $\Gamma\langle\Gamma'\rangle$ for the map where the bindings of $\Gamma'$ have overridden the ones from $\Gamma$.

**Remark 2** *Graphs are considered equal modulo removal of isolated nodes, and modulo the following rewriting rule:*

$$
N_1 \underset{\chi_2}{\overset{\chi_1}{\rightrightarrows}} N_2 \qquad - - \succ \qquad N_1 \xrightarrow{\chi_1 \wedge \chi_2} N_2 \qquad (1)
$$

*where $\wedge$ gives the most dangerous of two degrees:*

$$
\begin{array}{l}
\chi_1 \wedge \chi_2 = \text{☺} \text{ if } \chi_1 = \chi_2 = \text{☺} \\
\chi_1 \wedge \chi_2 = \text{☹} \text{ otherwise}
\end{array}
$$

The type system is defined by the inference rules given in figure 11.

The first rule T-Struct concerns the typing of basic structures $\langle \iota; o\rangle$. Given an input $I$ (which is arbitrary here, we do not consider type inference or type-checking issues) corresponding to $\iota$, and a type environment $\Gamma_o$ correponding to $o$, it checks that the definitions in $o$ indeed have the types mentioned in $\Gamma_o$.

The condition $\vdash \rightarrow_{\langle\iota;o\rangle}$ requires some explanation. We saw in section 3.2 that dependencies in an output are represented by its dependency graph $\rightarrow_o$. For structures (which are incomplete outputs), the corresponding notion is the concrete dependency graph. A concrete dependency graph is a graph over nodes. A node $N$ is an element of $\mathit{Nodes} = \mathit{Vars} \cup \mathit{Names}$. The dependency graph of a structure is defined in figure 14. It records dependencies in the structure (as was done for outputs), but takes external names into account when possible. Named definitions are represented by a name, and local definitions are represented by their variables. In order for types not to mention local components, we introduce a *lift* operation $\lfloor \rightarrow_{\langle\iota;o\rangle} \rfloor$, defined in figure 13. This lift operation replaces dependency paths that go through one or several local components by single edges labeled by the most dangerous degree appearing on the path. It then erases all local components. The result is an abstract dependency graph.

Finally, the rule checks that the imported types are well-formed, which would otherwise not be ensured, using the following notion of well-formedness.

**Definition 1 (Correct graphs)** *We say that a graph $\rightarrow$ is correct, and write $\vdash \rightarrow$, if and only if $\xrightarrow{\text{☹}}^+$ is an ordering on its nodes.*

$$
\frac{\vdash I \qquad \vdash O \qquad \vdash G \qquad \mathit{dom}(I) \perp \mathit{dom}(O) \qquad \mathit{Targets}(G) \subset \mathit{dom}(O)}{\vdash \langle I; O; G\rangle}
$$

$$
\frac{\vdash O}{\vdash \{O\}} \qquad \frac{\forall X \in \mathit{dom}(I) \vdash I(X)}{\vdash I}
$$

Figure 15: Well-formed types

| | |
|---|---|
| Lift | Transitive closure through local components |
| | $\dfrac{N_1 \xrightarrow{\chi_1} x \qquad x \xrightarrow{\chi_2}{}^{\square} N_2}{N_1 \xrightarrow{\chi_1 \wedge \chi_2}{}^{\square} N_2}$ $\qquad$ $\dfrac{N_1 \xrightarrow{\chi} N_2}{N_1 \xrightarrow{\chi}{}^{\square} N_2}$ |
| | Lift |
| | $\lfloor \rightarrow \rfloor = \rightarrow^{\square}_{\mid \mathit{Names} \times \mathit{Names}}$ |
| Sum | $G_1 + G_2 = G_1 \cup G_2$ |
| Freeze | $G \,!\, X = G\{X \leftarrow x\} \cup \{x \xrightarrow{\text{☹}} X\}$ ($x$ not mentioned in $G$) |
| Project | $G_{\mid \mathcal{N}} = G_{\mid \mathit{Names} \times \mathcal{N} \times \mathit{Degrees}}$ |
| Delete | $G_{\mid -\mathcal{N}} = G_{\setminus \mathit{Names} \times \mathcal{N} \times \mathit{Degrees}}$ |
| Hide | $G_{:-X_1 \ldots X_n} = G\{X_1 \mapsto x_1 \ldots X_n \mapsto x_n\}$ ($x_1 \ldots x_n$ fresh) |
| Show | $G_{:X_1 \ldots X_n} = G_{:- \mathit{Targets}(G) \setminus \{X_1 \ldots X_m\}}$ |
| Rename | $G\{r\} = \{(N_1\{r\}, N_2\{r\}, \chi) \mid (N_1, N_2, \chi) \in G\}$ |
| Split | $G_{X \succ Y} = (G \setminus G_{\mid \{. \rightarrow X\}}) \cup \{(Z, Y, \chi) \mid (Z, X, \chi) \in G\}$ |

Figure 13: Graph operations

$$
\frac{\chi = \mathit{Degree}(x', e) \qquad (L', x') \in \mathit{dom}(\langle\iota; o\rangle) \qquad (L[z^*] \triangleright x = e) \in o}{\mathit{Node}(L', x') \xrightarrow{\chi}_{\langle\iota;o\rangle} \mathit{Node}(L, x)}
$$

$$
\frac{(L_i, x_i) \in \mathit{dom}(\langle\iota; o\rangle) \qquad (L[x_1 \ldots x_n] \triangleright x = e) \in o}{\mathit{Node}(L_i, x_i) \xrightarrow{\text{☹}}_{\langle\iota;o\rangle} \mathit{Node}(L, x)}
$$

Figure 14: Dependencies in a structure

**Expressions:**

$$\frac{dom(\iota) = dom(I) \qquad \vdash I \qquad \vdash \Gamma_o \qquad \vdash \rightarrow_{\langle \iota;o \rangle} \qquad \Gamma\langle I \circ \iota^{-1} \uplus \Gamma_o \rangle \vdash o : \Gamma_o}{\Gamma \vdash \langle \iota; o \rangle : \langle I; \Gamma_o \circ \mathit{Input}(o); \lfloor \rightarrow_{\langle \iota;o \rangle} \rfloor \rangle} \quad \text{(T-Struct)}$$

$$\frac{I_1 \uplus O_1 \leftrightarrows I_2 \uplus O_2 \qquad \vdash G_1 \cup G_2 \qquad \Gamma \vdash e_1 : \langle I_1; O_1; G_1 \rangle \qquad \Gamma \vdash e_2 : \langle I_2; O_2; G_2 \rangle}{\Gamma \vdash e_1 + e_2 : \langle (I_1 \cup I_2) \setminus (O_1 \cup O_2); O_1 \uplus O_2; G_1 \cup G_2 \rangle} \quad \text{(T-Sum)}$$

$$\frac{\Gamma \vdash e : \langle I; O; G \rangle \qquad X \in dom(O)}{\Gamma \vdash e \,!\, X : \langle I; O; \lfloor G \,!\, X \rfloor \rangle} \quad \text{(T-Freeze)} \qquad \frac{\Gamma \vdash e : \langle \emptyset; O; G \rangle}{\Gamma \vdash \mathsf{close}\, e : \{O\}} \quad \text{(T-Close)}$$

$$\frac{\Gamma \vdash e : \langle I; O; G \rangle}{\Gamma \vdash e_{|X_1 \ldots X_n} : \langle I \uplus O_{\setminus \{X_1 \ldots X_n\}}; O_{| \{X_1 \ldots X_n\}}; G_{|\{X_1 \ldots X_n\}} \rangle} \quad \text{(T-Project)}$$

$$\frac{\Gamma \vdash e : \langle I; O; G \rangle}{\Gamma \vdash e_{|-X_1 \ldots X_n} : \langle I \uplus O_{| \{X_1 \ldots X_n\}}; O_{\setminus \{X_1 \ldots X_n\}}; G_{|-\{X_1 \ldots X_n\}} \rangle} \quad \text{(T-Delete)}$$

$$\frac{\Gamma \vdash e : \langle I; O; G \rangle \qquad \{X_1 \ldots X_n\} \subset dom(O)}{\Gamma \vdash e_{:-X_1 \ldots X_n} : \langle I; O_{\setminus \{X_1 \ldots X_n\}}; \lfloor G_{:-\{X_1 \ldots X_n\}} \rfloor \rangle} \quad \text{(T-Hide)} \qquad \frac{\Gamma \vdash e : \langle I; O; G \rangle \qquad \{X_1 \ldots X_n\} \subset dom(O)}{\Gamma \vdash e_{:X_1 \ldots X_n} : \langle I; O_{| \{X_1 \ldots X_n\}}; \lfloor G_{:\{X_1 \ldots X_n\}} \rfloor \rangle} \quad \text{(T-Show)}$$

$$\frac{\Gamma \vdash e : \langle I; O; G \rangle \qquad (cod(r) \setminus dom(r)) \perp (dom(I) \cup dom(O))}{\Gamma \vdash e[r] : \langle I\{r\}; O \circ \{r\}; G\{r\} \rangle} \quad \text{(T-Rename)}$$

$$\frac{\Gamma \vdash e : \langle I; O; G \rangle \qquad Y \notin dom(I) \cup dom(O)}{\Gamma \vdash e_{X \triangleright Y} : \langle I \uplus \{X : O(X)\}; O\{X \mapsto Y\}; G_{X \triangleright Y} \rangle} \quad \text{(T-Split)}$$

$$\frac{\forall i \in \{1 \ldots n\}, \Gamma \vdash e_i : M_i}{\Gamma \vdash \{X_1 = e_1 \ldots X_n = e_n\} : \{X_1 : M_1 \ldots X_n : M_n\}} \quad \text{(T-Record)} \qquad \frac{\Gamma \vdash e : \{O\}}{\Gamma \vdash e.X : O(X)} \quad \text{(T-Rselect)}$$

$$\frac{\vdash b \qquad \vdash \Gamma_b \qquad \Gamma\langle \Gamma_b \rangle \vdash b : \Gamma_b \qquad \Gamma\langle \Gamma_b \rangle \vdash e : M}{\Gamma \vdash \mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ e : M} \quad \text{(T-LetRec)} \qquad \frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \text{(T-Variable)}$$

**Sequences:**

$$\Gamma \vdash \epsilon : \emptyset \qquad \frac{\Gamma \vdash e : M \qquad \Gamma \vdash o : \Gamma_o}{\Gamma \vdash (L[x^*] \triangleright x = e, o) : \{x : M\} \uplus \Gamma_o} \qquad \frac{\Gamma \vdash e : M \qquad \Gamma \vdash b : \Gamma_b}{\Gamma \vdash (x = e, b) : \{x : M\} \uplus \Gamma_b}$$

Figure 11: Type system

**Definition 2 (Well-formed types)** *Figure 15 defines the sets of well-formed types, inputs, and outputs, as the least relation respecting the rules. A mixin type $\langle I; O; G\rangle$ must import and define disjoint sets of names, the targets of $G$ must be defined, and $G$ must be correct.*

The second rule T-Sum types the sum of two expressions. It verifies that names are bound to the same types in both expressions (using a $\rightleftarrows$ relation over types analogous to that over structures defined in figure 5), that the union of the two dependency graphs is still correct, and that names are not defined twice (i.e. are not part of both outputs). The result type shares the inputs, where defined names have been removed, and takes the union of the outputs and of the dependency graphs.

The third rule T-Freeze introduces a new operation $G!X \triangleright x$ on abstract graphs, which is again defined in figure 13. To freeze a name $X$, it first replaces $X$ with a fresh local variable $x$, making the graph temporarily non-abstract. Then, it adds a strict link from $x$ to $X$. This follows closely the semantics of freezing from figure 3, first making all other components call the local component $x$ instead of $X$, and then re-exporting $X$ as $x$ exactly. The link is forced to be a strict one by hypothesis 2.

The T-Close rule transforms a mixin type with no input into a record type. It looks very simple, but to prove it correct, we must show that well-ordered outputs yield well-ordered bindings by contraction rule Close.

The mixin projection rule T-Project, exactly like the corresponding contraction rule, keeps in the output types only the selected ones, transferring the other ones to the input types. The abstract graph is modified accordingly by the operation $G_{|\{X_1 \dots X_n\}}$, which removes the edges leading to unselected components. The T-Delete rule is the dual of rule T-Project.

The T-Hide rule removes the given names from the output. Additionally, it acts on the abstract graph $G$ as described in figure 13. It first replaces the given names by fresh variables, and then lifts the result, in order to obtain an abstract graph. Rule T-Show is its dual, as expected.

Rule T-Rename, given a mixin $e$ of type $\langle I; O; G\rangle$, deduces that $e$ renamed by $r$ has the same type, with input $I$ and output $O$ redirected to use the new names ($cod(r)$). Like the contraction rule Rename, it makes use of the $r_{\mathcal{N}}$ function, composed with $I$ and $O$. The abstract graph is renamed as well.

Given an expression $e$ of type $\langle I; O; G\rangle$, according to rule T-Split, the type of $e_{X \succ Y}$ is as follows. $X$ is added to the input, with the type it had in $O$. $X$ is renamed to $Y$ in the output. The graph $G$ is modified according to figure 13. $G_{|\{\cdot \to X\}}$ is the set of edges leading to $X$ in $G$. Basically, these edges are redirected to $Y$.

The T-Rselect and T-Record rules for typing record construction and selection are standard.

The T-LetRec for typing bindings $\mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ e$ is almost standard, except for its side-condition: the binding must be well-ordered with respect to its dependencies. The dependency graph of a binding $b$ is defined via the dependency graph of the equivalent output $Output(b) = Output(x_1 = e_1 \dots x_n = e_n) = (\_[\,] \triangleright x_1 = e_1 \dots \_[\,] \triangleright x_n = e_n)$. We define $\gtrdot_b$ by $\gtrdot_{Output(b)}$. A binding $b$ is said correct with respect to an ordering $>$ (written $>\vdash b$) if $>_b$ (the definition order in $b$) respects $>$ (in other words $>\subset>_b$). We abbreviate $\gtrdot_b \vdash b$ with $\vdash b$.

Finally, the typing of outputs and bindings is straightforward, since it consists in successively typing their definitions.

### 4.2 Soundness

In section 3, we presented *MM* with concrete, simple instances of *IsDefinedSize* and *Degree*. We now axiomatize the minimum conditions that they must satisfy.

**Hypothesis 1** *(Shape)*

- $x \notin Predictable$.
- $\langle \iota; o\rangle \in Predictable$ *and* $\{s_v\} \in Predictable$.
- *For all renamings* $\sigma$ *of variables,* $e\{\sigma\} \in Predictable$ *iff* $e \in Predictable$.
- *If* $\mathbb{E}[x] \in Predictable$, *then* $\mathbb{E}[v] \in Predictable$ *for all values* $v$.
- *If* $e \longrightarrow e'$ *and* $e \in Predictable$, *then* $e' \in Predictable$.
- *If* $e \in Predictable$ *and* $e' \in Predictable$, *then for any context* $\mathbb{E}$, $\mathbb{E}[e] \in Predictable$ *iff* $\mathbb{E}[e'] \in Predictable$.

We require the degree function to meet the following conditions.

**Hypothesis 2** *(Degree)*

- *If* $Degree(x, e) = \ominus$, *then* $e \in Predictable$.
- *If* $e \longrightarrow e'$ *and* $Degree(x, e) \neq \ominus$, *then* $Degree(x, e') \neq \ominus$.
- *If* $x \in FV(e) \setminus Capt_{\square}(\mathbb{E}[\mathbb{N}])$, *then* $Degree(x, \mathbb{E}[\mathbb{N}[e]]) = \ominus$.
- *If* $y \notin FV(v) \setminus Capt_{\square}(\mathbb{F})$, *then* $Degree(y, \mathbb{F}[v]) = Degree(y, \mathbb{F})$.
- *If for all* $x \in FV(e)$, $Degree(x, e') \leq Degree(x, e)$, *then for any context* $\mathbb{E}$ *and variable* $x \in FV(\mathbb{E}[e])$, $Degree(x, \mathbb{E}[e']) \leq Degree(x, \mathbb{E}[e])$.
- *For all* $x \notin dom(b)$ *and* $X \neq Y$ *and* $\chi \in \{\chi \mid X \xrightarrow{\chi}_{\langle X \triangleright x; o\rangle} N, o = (Output(b), Y \triangleright \_ = e)\}$, *we have* $Degree(x, \mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ e) \leq \chi$.

Under these two hypotheses, we can prove the soundness of the type system with respect to the reduction semantics. Detailed proofs are given in the companion technical report [17]; here, we just state the results.

**Lemma 1 (Subject reduction)** *If* $e \longrightarrow e'$ *and* $\Gamma \vdash e : M$, *then* $\Gamma \vdash e' : M$.

**Lemma 2 (Progress)** *If* $\Gamma \vdash e : M$ *and* $e$ *is not a result, then either* $e = \mathbb{E}[\mathbb{N}[x]]$ *with* $x$ *not captured by* $\mathbb{E}[\mathbb{N}]$, *or there exists* $e'$ *such that* $e \longrightarrow e'$.

**Theorem 1 (Soundness)** *The evaluation of a well-typed expression either does not terminate, or reaches a result, or gets stuck on a free variable.*

As free variables cannot appear during reduction, we have the following more standard corollary.

**Corollary 1 (Soundness for closed terms)** *The evaluation of a closed well-typed expression either does not terminate, or reaches a result.*

## 5 Related work

**Mixin-based inheritance** The notion of mixin originates in the object-oriented language Flavors [20], and was further investigated both as a linguistic device addressing many of the shortcomings of inheritance [12, 10] and as a semantic foundation for inheritance [7]. An issue with mixin classes that is generally not addressed is the treatment of instance fields and their initialization. Mixin classes where instance fields can be initialized by arbitrary expressions raise exactly the same problems of finding a correct evaluation order and detecting cyclic dependencies that we have addressed in this paper in the context of call-by-value mixin modules. Initialization can also be performed by an initialization method named `init` or some other conventional name, but this breaks data encapsulation.

As previously mentioned, the distinction between mixin modules (containing unevaluated computations) and modules (containing values), and the idea that only components of the latter can be accessed, is reminiscent of the distinction between classes and objects in class-based object-oriented languages.

**Language designs with mixin modules** Bracha [5] formulated the concept of mixin-based inheritance (composition) independently of an object-oriented setting. His mixins do not address the initialization issue. Duggan and Sourelis [8] extended his proposal and adapted it to ML. In their system, a mixin comprises a body, containing only function and datatype definitions, surrounded by a prelude and and an initialization section, containing arbitrary computations. During composition, only the bodies of the two mixins are connected, but neither the preludes nor the initialization sections. This prevents mixin composition from creating ill-founded recursive definitions. A distinctive feature of their system is that it features extensible datatypes and functions: composition merges the cases of similarly named functions and datatypes. However, all the expressive power of the system is concentrated in the single, complex composition operator, which contradicts Bracha's recommendation [5] for more atomic operators. We think that extensible functions and datatypes should rather be seen as an orthogonal feature. Further, there are some cases where one would like to interleave standard definitions and composable definitions.

Flatt and Felleisen [11] introduce the closely related concept of *units*, which adapt Bracha's ideas to Scheme and ML. A first difference with our proposal is that units do not directly feature late binding. Moreover, the initialization problem is handled differently. Their implementation of units for Scheme allows arbitrary computations within the definitions of unit components. The defined variables are implicitly initialized to `nil` before evaluating the right-hand sides of the defintions. Ill-founded definitions thus result either in a run-time type error (as in `let rec x = 1 + x`, since `nil` is not a valid argument to `+`) or in a value that is not a fixpoint of the recursive definition (as in `let rec x = cons(1,x)`, which binds `x` to `cons(1,nil)`). In contrast, our dynamic semantics always gets stuck when an ill-founded recursion appears during evaluation, and our type system statically prevents them from appearing. The formalization of units in [11, section 4] restricts definitions to syntactic values, but includes an initialization expression in each unit. This initialization expression can perform arbitrary computations and refer to the variables bound by the definitions, but is evaluated for its side-effects only. As in Duggan and Sourelis' system, this approach prevents the creation of ill-founded recursive definitions, but is less flexible than our approach.

**Linking calculi and mixin calculi** Cardelli [6] initiated the study of linking calculi. His system is a first-order linking model, that is, modules are compilation units and cannot be nested. His type system 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.

Ancona and Zucca [2] and Wells and Vestergaard [23] simultaneously introduced their calculi of mixin modules (*CMS* and **m**, respectively). *CMS* is parameterized over an arbitrary base language, has a call-by-name semantics, and is equipped with a sound type system. A drawback of *CMS* is that the type system does not restrict recursion at all, which breaks conservativity with respect to the base language. Indeed, assume that the base language forbids some recursive definitions (such as `x = y + 1 and y = x * 2`), which is the case for call-by-value languages. In the presence of mixin modules, such recursions can appear dynamically. Thus, access to other components of the same mixin module cannot be implemented directly, and an additional indirection must be added. Moreover, *CMS* is call-by-name, in the sense that the contents of modules are evaluated only at selection time. This is undesirable in the context of module systems for call-by-value languages.

The **m**-calculus handles evaluation inside modules, but the order of evaluation is unspecified, which is problematic in the presence of side-effects. Moreover, **m** does not have a type system, and does not rely on any base language. The focus is rather on the expressivity of the equational theory. Our *MM* language can be seen as a specialization of **m** to a call-by-value strategy, extended with automatic component reordering by default, user-definable order of evaluation, and a sound type system. Additionally, late binding is encodable in **m**, but it is not directly present, while it is in *MM*. Finally, **m** and *CMS* do not clearly separate mixin modules from modules.

Machkasova and Turbak [19] explore a linking calculus with a different equational theory than the one of **m**. The calculus is not confluent. Instead, it is argued that it is computationally sound, in the sense that all strategies lead to the same outcome. The system is untyped, and does not feature nested modules.

Hirschowitz and Leroy [15] adapt Ancona and Zucca's *CMS* to a call-by-value setting, in a way that is conservative with respect to the base language. They formalize a typed language $CMS_v$ of call-by-value mixin modules, whose semantics is given by a type-directed translation down to a call-by-value $\lambda$-calculus with let rec. The present paper improves on this earlier work by replacing the complex translation-based semantics by a more familiar source-level reduction semantics, and proving a more standard soundness theorem. Furthermore, the introduction of user-requested dependencies allows precise control on the order of evaluation.

## 6 Conclusion

We have presented a language of call-by-value mixin modules, equipped with a reduction semantics and a sound type

system. A companion paper [16] describes and proves the correctness of a compilation scheme for the let rec construct of *MM*.

Some open issues remain to be dealt with, which are related to different practical uses of mixin modules. If mixin modules are used as first-class, core language constructs, then the simple type system presented here is not expressive enough. Some form of polymorphism over mixin signatures seems necessary, along the lines of type systems for record concatenation proposed by Harper and Pierce [14] and by Pottier [21]. If one wants to build a module system based on mixin modules, then type abstraction and user-defined type components have to be considered. We are working on an extension of Leroy's module system [18] to mixin modules with type components.

Furthermore, in both cases, the programmer will probably have to write interfaces for mixin modules. In the type system presented here, the type of a mixin module must include its dependency graph. This is problematic for two reasons: first, it is cumbersome to write the whole graph of a mixin module; second, the dependency graph reveals too much information on the implementation of the mixin module, in the sense that small changes in the implementation must be reflected in the dependency graph, thus changing the interface. An issue that remains open is how to reduce the amount of dependency information that needs to be put in mixin interfaces, while still allowing static checking of well-formedness.

## References


[1] D. Ancona, S. Fagorzi, E. Moggi, and E. Zucca. Mixin modules and computational effects. Technical report, DISI, Università di Genova, 2002.

[2] D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Princ. and Practice of Decl. Prog.*, volume 1702 of *LNCS*, pages 62–79. Springer-Verlag, 1999.

[3] D. Ancona and E. Zucca. A calculus of module systems. *J. Func. Progr.*, 12(2):91–132, 2002.

[4] G. Boudol. The recursive record semantics of objects revisited. In D. Sands, editor, *Europ. Symp. on Progr.*, volume 2028 of *LNCS*, pages 269–283. Springer-Verlag, 2001.

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

[6] L. Cardelli. Program fragments, linking, and modularization. In *24th symp. Principles of Progr. Lang.*, pages 266–277. ACM Press, 1997.

[7] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Department of Computer Science, Brown University, 1989.

[8] D. Duggan and C. Sourelis. Mixin modules. In *Int. Conf. on Functional Progr.*, pages 262–273. ACM Press, 1996.

[9] L. Erkök, J. Launchbury, and A. Moran. Semantics of fixIO. In *Fixed Points in Comp. Sc.*, 2001.

[10] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Int. Conf. on Functional Progr.*, pages 94–104, 1998.

[11] M. Flatt and M. Felleisen. Units: cool modules for HOT languages. In *Prog. Lang. Design and Impl.*, pages 236–248. ACM Press, 1998.

[12] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *symp. Principles of Progr. Lang.*, pages 171–183. ACM Press, 1998.

[13] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symp. Principles of Progr. Lang.*, pages 123–137. ACM Press, 1994.

[14] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *symp. Principles of Progr. Lang.*, pages 131–142, Orlando, Florida, 1991.

[15] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, editor, *Europ. Symp. on Progr.*, volume 2305 of *LNCS*, pages 6–20, 2002.

[16] T. Hirschowitz, X. Leroy, and J. B. Wells. On the implementation of recursion in call-by-value functional languages. Research report RR-4728, INRIA, February 2003.

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

[18] X. Leroy. Manifest types, modules, and separate compilation. In *21st symp. Principles of Progr. Lang.*, pages 109–122. ACM Press, 1994.

[19] E. Machkasova and F. A. Turbak. A calculus for link-time compilation. In *Europ. Symp. on Progr.*, volume 1782 of *LNCS*, pages 260–274. Springer-Verlag, 2000.

[20] D. A. Moon. Object-oriented programming with Flavors. In *OOPSLA*, pages 1–8, 1986.

[21] F. Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, Nov. 2000.

[22] C. V. Russo. *Types for Modules*. PhD thesis, University of Edinburgh, 1998.

[23] J. B. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Europ. Symp. on Progr.*, volume 1782 of *LNCS*, pages 412–428. Springer-Verlag, 2000.