Call-by-Value Mixin Modules *

Reduction Semantics, Side Effects, Types

Tom Hirschowitz[1], Xavier Leroy[2], and J. B. Wells[3]

[1] École Normale Supérieure de Lyon
[2] INRIA Rocquencourt
[3] Heriot-Watt University

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

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 support for 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 [5].
- 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, method overriding and late binding answer the need for modifiability. However, viewed as a module system, classes have two weaknesses: they do not offer a general parameterization mechanism, and the mechanisms they offer to describe pre-computations (initialization of static and instance variables) lack generality. A module system should allow to naturally alternate function definitions with computational, possibly side-effective definitions using these functions.

Mixin modules [2] (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

* Partially supported by EPSRC grant GR/R 41545/01

initializations in mixins is still problematic. Most of the previous work on mixins, notably by Ancona and Zucca [1] and Wells and Vestergaard [23], is better suited to a call-by-name evaluation strategy. This strategy makes it impossible to trigger computations at initialization time.

Our goal in this paper is to define a call-by-value semantics for mixins that supports cleanly the evaluation of mixin components into values in a programmer-controlled evaluation order. In an earlier paper, Hirschowitz and Leroy [15] define a a typed language of mixins in a call-by-value setting, whose semantics is given by type-directed translation into enriched λ-calculus. The present paper improves over this first attempt in the following ways:

- Reduction semantics: We give a source-to-source, small-step reduction semantics for the mixin language. This semantics is simpler than the translation-based semantics, and is untyped. It also simplifies the proof of type soundness, which is now a standard argument by subject reduction and progress.
- Side effects: The semantics makes it easy for the programmer to (1) know when side-effects occur, and (2) control the order in which they occur.
- Anonymous definitions: Our system features anonymous definitions, that is, definitions that are evaluated, but not exported as components of the final module. The translation semantics first proposed by Hirschowitz and Leroy cannot handle anonymous definitions, because it is type-directed and anonymous definitions do not appear in mixin types.
- Practicality of mixin types: The type of a mixin must carry some dependency information about its contents. Requiring the dependency information to match exactly between the declared type of a mixin and its actual type, like Hirschowitz and Leroy did, is not practical. To address this issue, we introduce a new notion of subtyping w.r.t. dependencies, allowing a mixin to be viewed with more dependencies than it actually has. Furthermore, appropriate syntactic sugar allows to specify the dependencies of a large class of mixins with low syntactic overhead.

2 Overview

2.1 An operational semantics of mixins

Our central idea for bringing mutual recursion and modifiability to modules is to adapt the distinction between classes and objects to the context of mixins. Following this idea, this paper designs a kernel language of mixins called *MM*, which distinguishes mixins from actual modules. *Mixins* are dedicated to modularity operations, and feature parameterization, modifiability and mutual recursion. The contents of mixins are never reduced, so no computation takes place at the mixin level. *Modules* are dedicated to computation, contain fully evaluated values, and can be obtained by mixin instantiation, written close.

For the sake of simplicity, *MM* does not explicitly include a module construct. Instead, modules are encoded by a combination of records and value binding.

Roughly, a module struct $x_1 = e_1 \ldots x_n = e_n$ end is implemented by let rec $x_1 = e_1 \ldots x_n = e_n$ in $\{x_1 = e_1 \ldots x_n = e_n\}$. Based on previous work on value binding in a call-by-value setting [16], *MM* features a single value binding construct that expresses both recursive and non-recursive definitions. Basically, this construct evaluates the definitions from left to right, considering variables as values.

A mixin is a pair of a set of input variables $x_1 \ldots x_n$, and a list of output definitions $y_1 = e_1 \ldots y_m = e_m$, written $\langle x_1 \ldots x_n; y_1 = e_1 \ldots y_m = e_m \rangle$.

Mixins are equipped with operators adapted from previous works [2, 1, 23]. The main operator is composition: given two mixins e_1 and e_2, their composition $e_1 + e_2$ returns a new mixin whose output is the concatenation of those of e_1 and e_2, and whose inputs are the inputs of e_1 or e_2 that have not been filled by any output.

When closing a mixin $\langle \emptyset; y_1 = e_1 \ldots y_m = e_m \rangle$ without inputs (called *concrete*), the order in which to evaluate the definitions is not obvious. Indeed, the syntactic order arises from previous compositions, and does not necessarily reflect the intention of the programmer. For instance, the expression $\langle f; x = f\ 0 \rangle + \langle \emptyset; f = \lambda x.x + 1 \rangle$ evaluates to $\langle \emptyset; x = f\ 0, f = \lambda x.x + 1 \rangle$, which should be instantiated into struct $f = \lambda x.x + 1, x = f\ 0$ end (since definitions are evaluated from left to right). Thus, the close operator reorders definitions before evaluating them, thus turning a mixin into a module. It approximates an order in which the evaluation of each definition only needs previous definitions.

Unfortunately, this makes instantiation quite unintuitive in the presence of side effects. For example, if we are programming a ticket-vending machine for buying train tickets, it is reasonable to expect that the machine asks for the destination before asking whether the customer is a smoker or not. Indeed, the second question is useless if the trains to the requested destination are full. However, asking the second question does not require any information on the answer to the first one. So, if the program is built as an assembly of mixins, dependencies do not impose any order on the two questions, which can be a source of error. To handle this issue, our language of mixins provides programmer control over the order of evaluation: a definition can be annotated with the name of another one, to indicate that it should be evaluated after that one. For example, we can define $s = \langle \mathit{destination}; \mathit{smoker}[\mathit{destination}] = \ldots \rangle$. Intuitively, the annotation tells the system to do as if *smoker* depended on *destination*. This is why we call these annotations *fake dependencies*. Additionally, the system provides an operation for adding such dependencies *a posteriori*. For instance, assume our mixin was initially provided without the dependency annotation above: $s_0 = \langle \mathit{destination}; \mathit{smoker} = \ldots \rangle$. It is then important to be able to add it without modifying the source code. This is written $s_1 = s_{0\,\mathit{smoker}[\mathit{destination}]}$, which evaluates to the previous mixin s. Fake dependencies make *MM* ready for imperative features, although the formalization given in this paper does not include imperative features to keep it simpler.

2.2 Typing *MM*

The natural way to type-check mixins is via sets of type declarations for input and output components. For instance, let $m_1 = \langle x; y = e_1 \rangle$ and $m_2 = \langle y; x = e_2 \rangle$, where e_1 and e_2 denote two arbitrary expressions. It appears natural to give them the types $m_1 : \langle x : M_2; y : M_1 \rangle$ and $m_2 : \langle y : M_1; x : M_2 \rangle$, where M_1 and M_2 denote the types of e_1 and e_2, respectively, and the semi-colon separates the inputs from the outputs. The type of their composition is then $m : \langle \emptyset; x : M_2, y : M_1 \rangle$. While adequate for call-by-name mixins, this type system is not sound for call-by-value evaluation, because it does not guarantee that bindings generated at close time contain only well-founded recursive definitions that can be safely evaluated using call-by-value. In the example above, we could have x bound to $y+1$ and y bound to $x+1$, which is not well-founded. Yet, nothing in the type of m signals this problem.

In Sect. 4, we enrich these naive mixin types with dependency graphs describing the dependencies between definitions, and we formalize a simple (monomorphic) type system for *MM*. These graphs distinguish *strong dependencies*, which are forbidden in dependency cycles, from *weak dependencies*, which are allowed in dependency cycles. For instance, $x + 1$ strongly depends on x, while $\lambda y.x$ only weakly depends on it. The graphs are updated at each mixin operation, and allow to detect ill-founded recursions, while retaining most of the expressive power of *MM*.

Moreover, as mixin types carry dependency graphs, the types assigned to inputs may also contain graphs, and thus constrain the future mixins filling these inputs to have exactly the same graph. This policy is rather inflexible. To recover some flexibility, we introduce a notion of subtyping over dependency graphs: a mixin module with a dependency graph G can be viewed as having a more constraining graph. The type system of *MM* is the first to handle both subtyping over dependency graphs and anonymous definitions in mixins.

3 Syntax and dynamic semantics of *MM*

3.1 Syntax

We now formally define our kernel language of call-by-value mixin modules, called *MM*. Following Harper and Lillibridge [12], we distinguish *names* X from *variables* x. Variables are α-convertible, but names are not. *MM* expressions are defined in Fig. 1. Expressions include variables x, records (labeled by names) $\{X_1 = e_1 \ldots X_n = e_n\}$, and record selection $e.X$, which are standard.

The basic mixins are called *mixin structures*, which we abbreviate as simply *structures*. A structure $\langle \iota; o \rangle$ is a pair of an *input* ι of the shape $X_1 \triangleright x_1 \ldots X_n \triangleright x_n$, and of an *output* o of the shape $d_1 \ldots d_m$. The input ι maps external names imported by the structure to internal variables (used in o). The output o is an ordered list of *definitions* d. A definition is of the shape $L[x_1 \ldots x_n] \triangleright x = e$, where e is the *body* of the definition, and the *label* L is either a name X or the

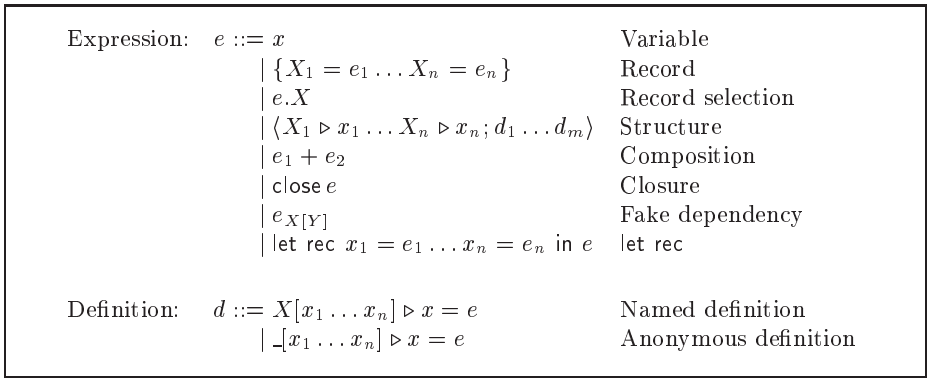

Fig. 1. Syntax of *MM*

anonymous label _. The possibly empty finite list of names $x_1 \ldots x_n$ is the list of *fake dependencies* of this definition on other definitions of the structure.

We provide four representative operators over mixins: compose $e_1 + e_2$, close $\mathsf{close}\, e$, delete $e_{|-X_1 \ldots X_n}$, and fake dependency $e_{X[Y]}$. Additional operators are formalized in Hirschowitz's PhD thesis [14].

Finally, *MM* features a single value binding construct $\mathsf{let\ rec}\ b\ \mathsf{in}\ e$ (where b is a list of recursive definitions $x_1 = e_1 \ldots x_n = e_n$, called a *binding*). As described in previous work [16], this construct encompasses ML recursive and non-recursive binding constructs. The binding construct of *MM* restricts recursion syntactically as follows.

(Backward dependencies) In a binding $b = (x_1 = e_1 \ldots x_n = e_n)$, we say that there is a *backward dependency* of x_i on x_j if $1 \leq i \leq j \leq n$ and $x_j \in FV(e_i)$, where $FV(e_i)$ denotes the set of free variables of e_i. A backward dependency of x_i on x_j is syntactically incorrect, except when e_j is of *predictable shape*.

(Predictable shape) Expressions of predictable shape are defined by $e_\downarrow \in \mathit{Predictable} ::= \{s_v\} \mid \langle \iota; o \rangle \mid \mathsf{let\ rec}\ b\ \mathsf{in}\ e_\downarrow$, where s_v ranges over evaluated record sequences (see below the definition of values).

In the following, we assume that all bindings are syntactically correct. Moreover, we consider expressions equivalent up to α-conversion of variables bound in structures and let rec expressions, and assume that no variable capture occurs. We also consider inputs equivalent up to reordering, and fake dependency lists equivalent up to reordering and repetition. Further, we assume that inputs, bindings, outputs, and structures (resp. inputs, records, outputs, and structures) do not define the same variable (resp. name) twice.

3.2 Semantics

Values and answers *MM* values are defined by

$$v ::= x \mid \{s_v\} \mid \langle X_1 \triangleright x_1 \ldots X_n \triangleright x_n ; d_1 \ldots d_n \rangle$$

where $s_v ::= X_1 = v_1 \dots X_1 = v_1$.

Evaluation answers are values, possibly surrounded by an evaluated binding: $a ::= v \mid \mathsf{let\ rec}\ b_v\ \mathsf{in}\ v$, where $b_v ::= x_1 = v_1 \dots x_n = v_n$.

Contraction relation In preparation for the reduction relation, we first define a local *contraction relation* $\rightsquigarrow$ by the rules in Fig. 2. Reduction will contain the closure of contraction under evaluation context.

Rule COMPOSE 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: ι is the union of ι_1 and ι_2, where names defined in o_1 or o_2 are removed. (In the rule, $\mathit{Input}(o)$ extracts an input $X \triangleright x$ from each named definition $X[y^*] \triangleright x = e$ in o.) The result output o is defined as the concatenation of o_1 and o_2. The side condition $\langle \iota_1; o_1 \rangle \asymp \langle \iota_2; o_2 \rangle$ means that the variables bound by one of the structures can only be mentioned by the other if they are tied to a common name in both structures. Lastly, o_1 and o_2 are required not to define the same names, by means of the function $\mathit{Names}(o_1)$ which denotes $\mathit{dom}(\mathit{Input}(o_1))$.

Rule CLOSE defines the instantiation of a structure $\langle \iota; o \rangle$. The input ι must be empty. The instantiation is in three steps.

- First, o is reordered according to its dependencies, to its fake dependencies, and to its syntactic ordering, thus yielding a new output $\overline{o}$. This is done by considering the syntactic definition order in o, written $\triangleright_o$, and the unlabeled dependency graph of o, written $\rightharpoonup_o$, which is defined by the two following inference rules

$$\frac{x' \in FV(e) \qquad (L[y^*] \triangleright x = e), (L'[z^*] \triangleright x' = e') \in o}{x' \rightharpoonup_o x} \qquad \frac{(L[x_1 \dots x_n] \triangleright x = e) \in o \qquad (L'[z^*] \triangleright x_i = e') \in o}{x_i \rightharpoonup_o x}$$

  Given an unlabeled graph $\rightharpoonup$ on variables, we define the binary relation $\rightarrowtail_{\rightharpoonup}$ by $\{x \rightarrowtail_{\rightharpoonup} y \mid x \rightharpoonup^+ y \text{ and } y \not\rightharpoonup^* x\}$. It defines a partial order on the variables defined by o, which respects dependencies, in the sense that if $x \rightarrowtail^+_{\rightharpoonup} y$, then x does not depend on y. The output $\overline{o}$ is then o, reordered w.r.t. the lexicographical order $(\rightarrowtail_{\rightharpoonup_o}, \triangleright_o)$. Thus, when dependencies do not impose an order, we choose the syntactic definition order as a default. By construction, in $\overline{o}$, all backward dependencies are part of cycles.
- Second, a binding $\mathit{Bind}(\overline{o})$ is generated, defining, for each definition $d = (L[y^*] \triangleright x = e)$ in $\overline{o}$, the definition $x = e$, in the same order as in $\overline{o}$. As we only write syntactically correct expressions, the rule has an implicit side condition that $\mathit{Bind}(\overline{o})$ be syntactically correct.
- Third, the values of the named definitions of $\overline{o}$ are grouped in a record $\mathit{Record}(\overline{o})$, with, for each named definition $X[y^*] \triangleright x = e$, a field $X = x$. This record is the result of the instantiation.

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 (in the rule, o is viewed as a finite map from pairs of a label and a variable to pairs of a finite set

Contraction rules

$$\frac{\langle \iota_1; o_1\rangle \Rightarrow \langle \iota_2; o_2\rangle \qquad \mathit{Names}(o_1) \perp \mathit{Names}(o_2)}{\langle \iota_1; o_1\rangle + \langle \iota_2; o_2\rangle \rightsquigarrow \langle (\iota_1 \cup \iota_2) \setminus \mathit{Input}(o_1, o_2); o_1, o_2\rangle} \quad (\textsc{Compose})$$

$$\mathsf{close}\langle \emptyset; o\rangle \rightsquigarrow \mathsf{let\ rec}\ \mathit{Bind}(\overline{o})\ \mathsf{in}\ \mathit{Record}(\overline{o}) \quad (\textsc{Close})$$

$$\langle \iota; o\rangle_{|-X_1 \ldots X_n} \rightsquigarrow \langle \iota + \mathit{Input}(o)_{|\,\{X_1 \ldots X_n\}}; o_{|\,dom(o)\setminus(\{X_1 \ldots X_n\} \times Vars)}\rangle \quad (\textsc{Delete})$$

$$\frac{(Y \triangleright y) \in dom(\langle \iota; o_1, X[z^*] \triangleright x = e, o_2\rangle)}{\langle \iota; o_1, X[z^*] \triangleright x = e, o_2\rangle_{X[Y]} \rightsquigarrow \langle \iota; o_1, X[yz^*] \triangleright x = e, o_2\rangle} \quad (\textsc{Fake})$$

$$\{s_v\}.X \rightsquigarrow s_v(X) \quad (\textsc{Select}) \qquad \frac{dom(b) \perp FV(\mathbb{L})}{\mathbb{L}[\mathsf{let\ rec}\ b\ \mathsf{in}\ e] \rightsquigarrow \mathsf{let\ rec}\ b\ \mathsf{in}\ \mathbb{L}[e]} \quad (\textsc{Lift})$$

Reduction rules

$$\frac{e \rightsquigarrow e'}{\mathbb{E}[e] \to \mathbb{E}[e']} \quad (\textsc{Context}) \qquad \frac{\mathbb{E}[\mathbb{D}](x) = v}{\mathbb{E}[\mathbb{D}[x]] \to \mathbb{E}[\mathbb{D}[v]]} \quad (\textsc{Subst})$$

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

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

Evaluation contexts

Evaluation context:
$\mathbb{E} ::= \mathbb{F} \mid \mathsf{let\ rec}\ b_v\ \mathsf{in}\ \mathbb{F} \mid \mathsf{let\ rec}\ \mathbb{B}[\mathbb{F}]\ \mathsf{in}\ e$
Lift context:
$\mathbb{L} ::= \{\mathbb{S}\} \mid \Box.X$
$\mid \mathsf{close}\,\Box \mid \Box_{|-X_1 \ldots X_n} \mid \Box_{X[Y]}$
$\mid \Box + e \mid v + \Box$
Dereferencing context:
$\mathbb{D} ::= \Box.X \mid \mathsf{close}\,\Box \mid \Box_{|-X_1 \ldots X_n} \mid \Box_{X[Y]}$
$\mid \Box + v_1 \mid v_2 + \Box$ (v_2 is not a variable).

Nested lift context:
$\mathbb{F} ::= \Box \mid \mathbb{L}[\mathbb{F}]$
Binding context:
$\mathbb{B} ::= b_v, x = \Box, b$
Record context:
$\mathbb{S} ::= s_v, X = \Box, s$

Access in evaluation contexts

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

Fig. 2. Dynamic semantics of *MM*

of variables and an expression). Second, the removed definitions remain bound as inputs, by adding the corresponding inputs to ι.

Rule FAKE describes the fake dependency operation, which allows to add a fake dependency to a mixin *a posteriori*. Let $dom(\langle \iota; o \rangle)$ denote $\iota + dom(o)$. Given a mixin $m = \langle \iota; o_1, X[z^*] \triangleright x = e, o_2 \rangle$, containing the name Y, bound by the variable y, the expression $m_{X[Y]}$ adds a fake dependency on y to the definition of X, thus yielding $\langle \iota; o_1, X[yz^*] \triangleright x = e, o_2 \rangle$.

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

Finally, in *MM*, there is no rule for eliminating let rec. Instead, evaluated bindings remain at top-level in the expression as a kind of run-time environment. Bindings that are not at top-level in the expression must be lifted before their evaluation can begin, as defined by rule LIFT and *lift contexts* $\mathbb{L}$.

Reduction relation We now define the dynamic semantics of *MM* by the global *reduction relation* $\rightarrow$, defined by the rules in Fig. 2.

As mentioned above, only the top-level binding can be evaluated. As soon as one of its definitions gets evaluated, evaluation can proceed with the next one, or with the enclosed expression if there is no definition left. This is enforced by the definition of *evaluation contexts* $\mathbb{E}$: evaluation happens under (if evaluated) or inside an optional top-level binding, and a *nested lift context* $\mathbb{F}$ (which is simply a series of lift contexts). If evaluation meets a binding inside the considered expression, then this binding is lifted to the top level of the expression, or just before the top-level binding if there is one. In this case, it is merged with the latter, either internally or externally, as described by rules IM and EM, respectively. External and internal substitutions (rules SUBST, EA and IA) allow to copy one of the already evaluated definitions of the top-level binding, when they are needed by the evaluation, i.e. when they appear in a *dereferencing context*. The condition that v_2 is not a variable in the grammar ensures determinism of the reduction in cases such as $x + y$. The left argument is always copied first.

Finally, rule CONTEXT extends contraction to evaluation contexts.

4 Static semantics of *MM*

Types are defined by $M \in \mathit{Types} ::= \{O\} \mid \langle I; O; G \rangle$, where I and O are *signatures*, that is, finite maps from names to types, and where G is a graph over names, labeled by *degrees*. A degree χ is one of ☹ and ☺, respectively representing strong and weak dependencies. There are only two kinds of types: record types $\{O\}$ and mixin types $\langle I; O; G \rangle$. *Environments* Γ are finite maps from variables to types.

The type system is defined in Fig. 3. After the standard typing rule T-VARIABLE for variables, rule T-STRUCT defines the typing of structures $\langle \iota; o \rangle$. The rule has to guess a well-formed input signature I corresponding to ι, and a well-formed type environment Γ_o corresponding to o. Type, signature, and environment well-formedness only requires that for any mixin type $\langle I; O; G \rangle$,

Expressions

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \text{(T-Variable)}$$

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

$$\frac{\Gamma \vdash e : M' \qquad M' \leq M \qquad \vdash M}{\Gamma \vdash e : M} \quad \text{(T-Sub)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : \langle I_1 ; O_1 ; G_1 \rangle \qquad \Gamma \vdash e_2 : \langle I_2 ; O_2 ; G_2 \rangle \\ \vdash G_1 \cup G_2 \qquad dom(I) = dom(I_1) \cup dom(I_2) \qquad \vdash I \\ I_{\mid dom(I_1)} \leq I_1 \qquad I_{\mid dom(I_2)} \leq I_2 \qquad (O_1 + O_2) \leq I_{\mid dom(O_1 + O_2)} \end{array}}{\Gamma \vdash e_1 + e_2 : \langle I ; O_1 + O_2 ; G_1 \cup G_2 \rangle} \quad \text{(T-Compose)}$$

$$\frac{\Gamma \vdash e : \langle I ; O ; G \rangle \qquad dom(I) = dom(O)}{\Gamma \vdash \mathsf{close}\, e : \{O\}} \quad \text{(T-Close)}$$

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

$$\frac{\Gamma \vdash e : \langle I ; O ; G \rangle \qquad X \in dom(O) \qquad Y \in dom(I) \qquad \vdash G_{X[Y]}}{\Gamma \vdash e_{X[Y]} : \langle I ; O ; G_{X[Y]} \rangle} \quad \text{(T-Fake)}$$

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

$$\frac{\forall X \in dom(s), \Gamma \vdash s(X) : O(X)}{\Gamma \vdash \{s\} : \{O\}} \quad \text{(T-Record)} \qquad \frac{\Gamma \vdash e : \{O\}}{\Gamma \vdash e.X : O(X)} \quad \text{(T-Select)}$$

Sequences

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

$$\frac{\Gamma \vdash e : M \qquad \Gamma \vdash b : \Gamma_b}{\Gamma \vdash (x = e, b) : \{x : M\} + \Gamma_b}$$

Fig. 3. Static semantics of *MM*

G is *safe*, in the sense that its cycles only contain weak dependencies (labeled by ☺), and $O \leq I_{|\,dom(O)}$, in the sense of signature subtyping, defined below. Given I and Γ_o, the rule checks that the definitions in o indeed have the types mentioned in Γ_o. The types of named definitions of o, obtained by composing Γ_o with $Input(o)$, are retained both as inputs and outputs. Finally, the condition $\vdash \rightarrow_{\langle \iota;o \rangle}$ checks that the dependencies of the structure are safe. It relies on the labeled dependency graph of $\langle \iota; o \rangle$, which is defined by the two following inference rules.

$$\frac{(L', x') \in dom(\langle \iota; o \rangle) \quad (L[y^*] \triangleright x = e) \in o}{Node(L', x') \xrightarrow{Degree(x', e)}_{\langle \iota;o \rangle} Node(L, x)} \qquad \frac{(L_i, x_i) \in dom(\langle \iota; o \rangle) \quad (L[x_1 \ldots x_n] \triangleright x = e) \in o}{Node(L_i, x_i) \xrightarrow{☺}_{\langle \iota;o \rangle} Node(L, x)}$$

where $Node(L, x)$ denotes L if L is a name, and x otherwise. The edges of this graph are labeled by degrees, which are computed by the *Degree* function, defined if $x \in FV(e)$ by $Degree(x, e) = ☺$ if $e \in Predictable$ and $Degree(x, e) = ☹$ otherwise. Finally, variables should not appear in types, so we *lift* the graph to a labeled graph over names written $\lfloor \rightarrow_{\langle \iota;o \rangle} \rfloor$. Namely, we extend edges through anonymous components: for each path $N_1 \xrightarrow{\chi_1} x \xrightarrow{\chi_2} N_2$, we add the edge $N_1 \xrightarrow{\chi} N_2$, where χ is the minimum of χ_1 and χ_2, given that $☹ < ☺$. Then, $\lfloor \rightarrow_{\langle \iota;o \rangle} \rfloor$ denotes the restriction of the resulting graph to names.

The subsumption rule T-Sub materializes the presence of subtyping in *MM*. Subtyping is defined by the following two rules:

$$\frac{I_2 \leq I_1 \qquad O_1 \leq O_2 \qquad G_1 \subset G_2}{\langle I_1; O_1; G_1 \rangle \leq \langle I_2; O_2; G_2 \rangle} \qquad \frac{O_1 \leq O_2}{\{O_1\} \leq \{O_2\}}$$

where subtyping between signatures is defined component-wise. Subtyping allows a dependency graph to be replaced by a more constraining graph. Section 5 illustrates the practical importance of subtyping between dependency graphs.

Rule T-Compose types the composition of two expressions. It guesses a lower bound I of the input signatures I_1 and I_2 of its arguments, such that $dom(I) = dom(I_1) \cup dom(I_2)$. This lower bound is used as the input signature of the result. Checking that it is a lower bound implies that common names between I_1 and I_2 have compatible types. The rule also checks that the union of the two dependency graphs is safe, and that no name is defined twice (i.e. is not in both outputs). The result type shares the inputs and takes the union of the outputs and of the dependency graphs.

Rule T-Close transforms a mixin type whose inputs are all matched by its outputs into a record type.

Rule T-Delete, exactly as the corresponding contraction rule, removes the selected names from the output types, reporting the other ones in the input signature. The abstract graph is modified accordingly by the operation $G_{|-\{X_1 \ldots X_n\}}$, which removes the edges leading to the deleted components.

Rule T-Fake types an expression of the shape $e_{X[Y]}$. If e has a type $\langle I; O; G \rangle$, with $X \in dom(O)$, and $Y \in dom(I)$, then adding a fake dependency of X on

Y only modifies the graph G: $G_{X[Y]}$ denotes G, augmented with a strong edge from Y to X. The rule checks that this does not make the graph unsafe.

Rule T-LetRec for typing bindings $\mathsf{let\ rec}\ b\ \mathsf{in}\ e$ is standard, except for its side condition: $\vdash b$ means that b does not contain backward dependencies on definitions of unpredictable shape, and is well ordered with respect to its dependencies, in the following sense. The dependency graph $\rightarrow_b$ of $b = (x_1 = e_1 \ldots x_n = e_n)$ is defined as the labeled dependency graph of the equivalent output $(\_[] \triangleright x_1 = e_1 \ldots \_[] \triangleright x_n = e_n)$. Then, we require that all paths of $\rightarrow_b$ whose last edge is labeled by $\odot$ are forward. This is sufficient to ensure that b contains no dependency problem.

The T-Select and T-Record rules for typing record construction and selection are standard. Rule T-Select has an implicit side-condition that $X \in dom(O)$.

Finally, Fig. 3 also presents the typing of sequences, outputs and bindings, which is straightforward, since it consists in successively typing their definitions.

Theorem 1 (Soundness) *A closed, well-typed expression can either not terminate or reach an answer.*

The proof of this theorem (via the standard subject reduction and progress properties) can be found in Hirschowitz's PhD thesis [14].

5 Practical syntactic signatures and subtyping w.r.t. dependencies

As mentioned in the introduction, enriching mixin types with dependency graphs without graph subtyping would make the type system too rigid. Assuming such a system, consider a mixin e which imports a mixin X. The type of e has an input declaration named X that associates a graph to X. If we later want to use e twice in the program, composing it with two different mixins e' and e'', it is unlikely that X has exactly the same dependency graph in e' and e'', so we cannot attribute a graph to X in e that allows both compositions. Furthermore, from the standpoint of separate development, the dependency graph is part of the specification of a mixin. It informs clients of dependencies, but also of non-dependencies. Thus, definitions must depend exactly on the components that the graph claims they depend on. So, if the implementation of a mixin changes for any reason such as optimization, bug fix, etc, then probably its specification will also have to change. This is undesirable for separate development, which encourages the independent development of mixins, based on stable specifications.

Our previous type systems for mixins [15, 17] suffer from this drawback: they require the dependency graph of an output to exactly match the one of the input it fills. We improve over these type systems here, by incorporating a simple notion of subtyping in our type system for *MM*, which allows to see a mixin with dependency graph G as a mixin with a more constraining dependency graph, that is, a super graph of G. The idea is that when giving the type of an input, the

programmer (or possibly a type inference algorithm, although we have no such algorithm to propose yet) chooses a reasonably constraining dependency graph that remains compatible with the uses made of the input. Subtyping, then, allows the input to be filled by less constrained definitions.

$$
\begin{array}{ll}
[\![\mathsf{mixsig}\ \epsilon\ \mathsf{end}, M]\!] & = M \\
[\![\mathsf{mixsig}\ ?X : M, Q_1 \dots Q_n\ \mathsf{end}, \langle I; O; G\rangle]\!] & = [\![\mathsf{mixsig}\ Q_1 \dots Q_n\ \mathsf{end}, \\
 & \quad \langle I + \{X : M\}; O; G\rangle]\!] \\
[\![\mathsf{mixsig}\ !X : M, Q_1 \dots Q_n\ \mathsf{end}, \langle I; O; G\rangle]\!] & = [\![\mathsf{mixsig}\ Q_1 \dots Q_n\ \mathsf{end}, \\
 & \quad \langle I + \{X : M\}; O + \{X : M\}; \\
 & \quad G \cup \{Y \xrightarrow{\text{☹}} X \mid Y \in dom(I)\}\rangle]\!] \\
[\![\,\mathsf{mixsig} & = [\![\mathsf{mixsig}\ Q_1 \dots Q_n\ \mathsf{end}, \\
\quad [(!X_1 : M_1) \dots (!X_p : M_p) & \quad \langle I + \{X_1 : M_1 \dots X_m : M_m\}; \\
\quad (?X_{p+1} : M_{p+1}) \dots (?X_m : M_m)], & \quad O + \{X_1 : M_1 \dots X_p : M_p\}; \\
\quad Q_1 \dots Q_n & \quad G \cup \bigcup_{i \in \{1 \dots p\}} \{Y \xrightarrow{\text{☹}} X_i \mid Y \in dom(I)\} \\
\quad \mathsf{end}, & \quad \cup \bigcup_{i \in \{1 \dots p\}, j \in \{1 \dots m\}} \{X_j \xrightarrow{\text{☺}} X_i\}\rangle]\!] \\
\quad \langle I; O; G\rangle]\!] &
\end{array}
$$

Fig. 4. Syntactic sugar for writing graphs

Another related problem is that dependency graphs, and *a fortiori* the constraining graphs mentioned above, are very cumbersome to write by hand for the programmer. To alleviate this issue, we propose the introduction of appropriate syntactic sugar. Our idea is to add a form of mixin type $\mathsf{mixsig}\ Q_1 \dots Q_n\ \mathsf{end}$, with

$$
\begin{array}{l}
Q ::= U \mid [U_1 \dots U_n] \\
U ::= ?X : M \mid !X : M
\end{array}
$$

This new construct is a list of enriched specifications Q. An enriched specification Q is either a single declaration U, or a block of single declarations $[U_1 \dots U_n]$. A single declaration assigns a type to a name, and has a flag ? or !, to indicate that it is an input or an output, respectively. Blocks are considered equivalent modulo the order, and they represent groups of potentially recursive definitions of predictable shape. Single definitions alone represent computations of any shape.

This construct can be elaborated to core *MM* types, as defined in Fig. 4. Basically, ? declarations are inputs, and ! declarations are both inputs and outputs. A single ! declaration, is considered as strongly depending on all the preceding declarations. A ! declaration in a block is considered to strongly depend on the preceding declarations, and to weakly depend on all the declarations of its block.

Our syntactic sugar allows to write mixin types almost like module types, thus making them more practical.

6 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 [11, 9] and as a semantic foundation for inheritance [4]. Here, we call this kind of mixins *mixin classes*. 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 mixins. Initialization can also be performed by an initialization method with a standard name (say, `init`), but this breaks data encapsulation.

Recursive modules Harper et al. [5, 7] and Russo [22] extend the ML module system with recursive definitions of modules. This addresses the mutual recursion issue we mentioned in introduction, but not the modifiability (open recursion) issue. Russo relies on lazy evaluation for the recursive definitions and makes no attempt to statically detect ill-founded recursions. Harper et al. use a standard call-by-value fixed-point operator, and statically constrain components of recursively-defined modules to be *valuable*. This is less flexible than our proposal, since module components can only weakly depend on the recursive variable. Recent work by Dreyer [6] lifts this restriction by using an effect system to track strong dependencies on recursively-defined variables.

Language designs with mixins Bracha [2] 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 data-type definitions, surrounded by a prelude 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 ensures that mixin composition never creates ill-founded recursive definitions, but prevents interleaving between standard definitions and composable definitions.

Flatt and Felleisen [10] introduce the closely related concept of *units*. A first difference with our proposal is that units do not feature late binding. Moreover, the initialization problem is handled differently. The formalization of units in [10, Sect. 4] restricts definitions to syntactic values, but includes in each unit an initialization expression that can perform arbitrary computations. Like Duggan and Sourelis's approach, this approach prevents the creation of ill-founded recursive definitions, but is less flexible than our approach. The 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 definitions and updating the defined variables with the results of the computation. Ill-founded recursions are thus not

prevented statically, and result either in a run-time type error or in a value that is not a fixed-point of the recursive definition.

Linking calculi and mixin calculi Cardelli [3] 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.

Machkasova and Turbak [19] explore a very expressive linking calculus, which 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.

Ancona and Zucca [1] propose a call-by-name module system called *CMS*. As *MM*, *CMS* extends Jigsaw by allowing any kind of expressions as mixin definitions, not just values. Unlike in *MM*, in *CMS*, there is no distinction between modules and mixin modules: in call-by-name languages, the contents of modules are not evaluated until selection, so it makes sense to avoid the distinction. In a call-by-value setting, the contents of a module are eagerly evaluated, so *CMS* does not model call-by-value modules. From the standpoint of typing, *CMS* is quite close to *MM*, except that it does not control recursive definitions. This is consistent with most call-by-name languages, which generally loop or raise an exception in case of ill-founded definitions.

As *CMS*, Wells and Vestergaard's **m**-calculus [23] is targeted to call-by-name evaluation. Nevertheless, it has a rich equational theory that allows to see *MM* as a specialization of **m** to call-by-value plus built-in late binding behavior (encoded in **m**), explicit distinction between mixins and modules, programmer control over the order of evaluation, and a sound and flexible type system.

7 Conclusion

We have presented a language of call-by-value mixin modules, equipped with a reduction semantics and a sound type system. 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 module types seems necessary, along the lines of type systems for record concatenation proposed by Harper and Pierce [13] 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 extending the type systems for ML modules [18, 12] to mixin modules with type components.

References

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

2. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
3. L. Cardelli. Program fragments, linking, and modularization. In *24th symp. Principles of Progr. Lang.*, pages 266–277. ACM Press, 1997.
4. W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Department of Computer Science, Brown University, 1989.
5. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Prog. Lang. Design and Impl.*, pages 50–63. ACM Press, 1999.
6. D. Dreyer. A type system for well-founded recursion. In *31st symp. Principles of Progr. Lang.* ACM Press, 2004. To appear.
7. D. R. Dreyer, R. Harper, and K. Crary. Towards a practical type theory for recursive modules. Technical Report CMU-CS-01-112, Carnegie Mellon University, Pittsburgh, PA, Mar. 2001.
8. D. Duggan and C. Sourelis. Mixin modules. In *Int. Conf. on Functional Progr.*, pages 262–273. ACM Press, 1996.
9. R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Int. Conf. on Functional Progr.*, pages 94–104, 1998.
10. M. Flatt and M. Felleisen. Units: cool modules for HOT languages. In *Prog. Lang. Design and Impl.*, pages 236–248. ACM Press, 1998.
11. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *25th symp. Principles of Progr. Lang.*, pages 171–183. ACM Press, 1998.
12. 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.
13. R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *18th symp. Principles of Progr. Lang.*, pages 131–142, Orlando, Florida, 1991.
14. T. Hirschowitz. *Mixin modules, modules, and extended value binding in a call-by-value setting*. PhD thesis, University of Paris VII, Dec. 2003. Preliminary version available on the Web, `http://pauillac.inria.fr/~hirschow/phd/state.html`.
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. Compilation of extended recursion in call-by-value functional languages. In *Princ. and Practice of Decl. Prog.*, pages 160–171. ACM Press, 2003.
17. T. Hirschowitz, X. Leroy, and J. B. Wells. A reduction semantics for call-by-value mixin modules. Research report RR-4682, INRIA, Jan. 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. Recursive structures for Standard ML. In *Int. Conf. on Functional Progr.*, pages 50–61, 2001.
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.