Mixin modules in a call-by-value setting

Tom Hirschowitz and Xavier Leroy

INRIA Rocquencourt
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France
{Tom.Hirschowitz,Xavier.Leroy}@inria.fr

Abstract. The ML module system provides powerful parameterization facilities, but lacks the ability to split mutually recursive definitions across modules, and does not provide enough facilities for incremental programming. A promising approach to solve these issues is Ancona and Zucca's mixin modules calculus *CMS*. However, the straightforward way to adapt it to ML fails, because it allows arbitrary recursive definitions to appear at any time, which ML does not support. In this paper, we enrich *CMS* with a refined type system that controls recursive definitions through the use of dependency graphs. We then develop a separate compilation scheme, directed by dependency graphs, that translate mixin modules down to a CBV λ-calculus extended with a non-standard `let rec` construct.

1 Introduction

Modular programming and code reuse are easier if the programming language provides adequate features to support them. Three important such features 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 [9], 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.

The concept of *mixins*, first introduced as a generalization of inheritance in class-based OO languages [8], then extended to a family of module systems [13, 3, 15, 21], offers a promising and elegant solution to this problem. A mixin is a collection of named components, either defined (bound to a definition) or deferred (declared without definition). The basic operation on mixins is the sum, which takes two mixins and connects the defined components of one with the similarly-named deferred components of the other; this provides natural support for cross-mixin recursion. A mixin is named and can be summed several times with different mixins; this allows powerful parameterization, including but not

restricted to an encoding of ML functors. Finally, the mixin calculus of Ancona and Zucca [3] supports both late binding and early binding of defined components, along with deleting and renaming operations, thus providing excellent support for incremental programming.

Our long-term goal is to extend the ML module system with mixins, taking Ancona and Zucca's *CMS* calculus [3] as a starting point. There are two main issues: one, which we leave for future work, is to support type components in mixins; the other, which we address in this paper, is to equip *CMS* with a call-by-value semantics consistent with that of the core ML language. Shifting *CMS* from its original call-by-name semantics to a call-by-value semantics requires a precise control of recursive definitions created by mixin composition. The call-by-name semantics of *CMS* puts no restrictions on recursive definitions, allowing ill-founded ones such as `let rec x = 2 * y and y = x + 1`, causing the program to diverge when `x` or `y` is selected. In an ML-like, call-by-value setting, recursive definitions are statically restricted to syntactic values, *e.g.* `let rec f =` λ`x... and g =` λ`y...` This provides stronger guarantees (ill-founded recursions are detected at compile-time rather than at run-time), and supports more efficient compilation of recursive definitions. Extending these two desirable properties to mixin modules in the presence of separate compilation [9, 18] is challenging: illegal recursive definitions can appear a posteriori when we take the sum `A + B` of two mixin modules, at a time where only the signatures of `A` and `B` are known, but not their implementations.

The solution we develop here is to enrich the *CMS* type system, adding graphs in mixin signatures to represent the dependencies between the components. The resulting typed calculus, called CMS_v, guarantees that recursive definitions created by mixin composition evaluate correctly under a call-by-value regime, yet leaves considerable flexibility in composing mixins. We then provide a type-directed, separate compilation scheme for CMS_v. The target of this compositional translation is λ_B, a simple call-by-value λ-calculus with a non-standard `let rec` construct in the style of Boudol [6]. Finally, we prove that the compilation of a type-correct CMS_v mixin is well typed in a sound, non-standard type system for λ_B that generalizes that of [6], thus establishing the soundness of our approach.

The remainder of the paper is organized as follows. Section 2 gives a high-level overview of the *CMS* and CMS_v mixin calculi, and explains the recursion problem. Section 3 defines the syntax and typing rules for CMS_v, our call-by-value mixin module calculus. The compilation scheme (from CMS_v to λ_B) is presented in section 4. Section 5 outlines a proof of correctness of the compilation scheme. We review related work in section 6, and conclude in section 7. Proofs are omitted, but can be found in the long version of this paper [17].

2 Overview

2.1 The *CMS* calculus of mixins

We start this paper by an overview of the *CMS* module calculus of [3], using an ML-like syntax for readability. A basic mixin module is similar to an ML structure, but may contain "holes":

```
mixin Even =  mix
  ? val odd: int -> bool             (* odd is deferred *)
  let even = λx. x = 0 or odd(x-1)   (* even is defined  *)
end
```

In other terms, a mixin module consists of defined components, `let`-bound to an expression, and deferred components, declared but not yet defined. The fundamental operator on mixin modules is the sum, which combines the components of two mixins, connecting defined and deferred components having the same names. For example, if we define `Odd` as

```
mixin Odd = mix
  ? val even: int -> bool
  let odd = λx. x > 0 and even(x-1)
end
```

the result of `mixin Nat = Even + Odd` is equivalent to writing

```
mixin Nat = mix
  let even = λx. x = 0 or odd(x-1)
  let odd  = λx. x > 0 and even(x-1)
end
```

As in class-based languages, all defined components of a mixin are mutually recursive by default; thus, the above should be read as the ML structure

```
module Nat = struct
  let rec even = λx. x = 0 or odd(x-1)
      and odd  = λx. x > 0 and even(x-1)
end
```

Another commonality with classes is that defined components are late bound by default: the definition of a component can be overridden later, and other definitions that refer to this component will "see" the new definition. The overriding is achieved in two steps: first, deleting the component via the \ operator, then redefining it via a sum. For instance,

```
mixin Nat' = (Nat \ even) + (mix let even = λx. x mod 2 = 0 end)
```

is equivalent to the direct definition

```
mixin Nat' = mix
  let even = λx. x mod 2 = 0
  let odd  = λx. x > 0 and even(x-1)
end
```

Early binding (definite binding of a defined name to an expression in all other components that refer to this name) can be achieved via the freeze operator !. For instance, `Nat ! odd` is equivalent to

```
mix
  let even = let odd = λx. x > 0 and even(x-1) in
             λx. x = 0 or odd(x-1)
  let odd  = λx. x > 0 and even(x-1)
end
```

For convenience, our CMS_v calculus also provides a `close` operator that freezes all components of a mixin in one step. Projections (extracting the value of a mixin component) are restricted to closed mixins – for the same reasons that in class-based languages, one cannot invoke a method directly from a class: an instance of the class must first be taken using the "new" operator.

A component of a mixin can itself be a mixin module. Not only does this provide ML-style nested mixins, but it also supports a general encoding of ML functors, as shown in [2].

2.2 Controlling recursive definitions

As recalled in introduction, call-by-value evaluation of recursive definitions is usually allowed only if the right-hand sides are syntactic values (*e.g.* λ-abstractions or constants). This semantic issue is exacerbated by mixin modules, which are in essence big mutual `let rec` definitions. Worse, ill-founded recursive definitions can appear not only when defining a basic mixin such as

```
mixin Bad = close(mix  let x = y + 1   let y = x * 2  end)
```

but also *a posteriori* when combining two innocuous-looking mixins:

```
mixin OK1 = mix  ? val y : int   let x = y + 1  end
mixin OK2 = mix  ? val x : int   let y = x * 2  end
mixin Bad = close(OK1 + OK2)
```

Although `OK1` and `OK2` contain no ill-founded recursions, the sum `OK1 + OK2` contains one. If the definitions of `OK1` and `OK2` are known when we type-check and compile their sum, we can simply expand `OK1 + OK2` into an equivalent monolithic mixin and reject the faulty recursion. But in a separate compilation setting, `OK1 + OK2` can be compiled in a context where the definitions of `OK1` and `OK2` are not known, but only their signatures are. Then, the ill-founded recursion cannot be detected. This is the major problem we face in extending ML with mixin modules.

One partial solution, that we find unsatisfactory, is to rely on lazy evaluation to implement a call-by-name semantics for modules, evaluating components only at selection or when the module is closed. (This is the approach followed by the Moscow ML recursive modules [20], and also by class initialization in Java.) This would have several drawbacks. Besides potential efficiency problems, lazy evaluation does not mix well with ML, which is a call-by-value imperative language. For instance, ML modules may contain side-effecting initialization code

that must be evaluated at predictable program points; that would not be the case with lazy evaluation of modules. The second drawback is that ill-founded recursive definitions (as in the `Bad` example above) would not be detected statically, but cause the program to loop or fail at run-time. We believe this seriously decreases program safety: compile-time detection of ill-founded recursive definitions is much preferable.

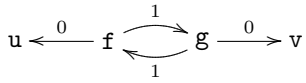
Our approach consists in enriching mixin signatures with graphs representing the dependencies between components of a mixin module, and rely on these graphs to detect statically ill-founded recursive definitions. For example, the `Nat` and `Bad` mixins shown above have the following dependency graphs:

Edges labeled 0 represent an immediate dependency: the value of the source node is needed to compute that of the target node. Edges labeled 1 represent a delayed dependency, occurring under at least one λ-abstraction; thus, the value of the target node can be computed without knowing that of the source node. Ill-founded recursions manifest themselves as cycles in the dependency graph involving at least one "0" edge. Thus, the correctness criterion for a mixin is, simply: all cycles in its dependency graph must be composed of "1" edges only. Hence, `Nat` is correct, while `Bad` is rejected.

The power of dependency graphs becomes more apparent when we consider mixins that combine recursive definitions of functions and immediate computations that sit outside the recursion:

```
mixin M1 = mix                  mixin M2 = mix
  ? val g : ...                   ? val f : ...
  let f = λx. ...g...             let g = λx. ...f...
  let u = f 0                     let v = g 1
end                             end
```

The dependency graph for the sum `M1 + M2` is:



It satisfies the correctness criterion, thus accepting this definition; other systems that record a global "valuability" flag on each signature, such as the recursive modules of [11], would reject this definition.

3 The CMS_v calculus

We now define formally the syntax and typing rules of CMS_v, our call-by-value variant of *CMS*.

Core terms:	$C ::= x \mid cst$	variable, constant
	$\mid \lambda x.C \mid C_1\ C_2$	abstraction, application
	$\mid E.X$	component projection
Mixin terms:	$E ::= C$	core term
	$\mid \langle \iota; o \rangle$	mixin structure
	$\mid E_1 + E_2$	sum
	$\mid E[X \leftarrow Y]$	rename X to Y
	$\mid E \,!\, X$	freeze X
	$\mid E \setminus X$	delete X
	$\mid \texttt{close}(E)$	close
Input assignments:	$\iota ::= x_i \overset{i \in I}{\mapsto} X_i$	ι injective
Output assignments:	$o ::= X_i \overset{i \in I}{\mapsto} E_i$	
Core types:	$\tau ::= \texttt{int} \mid \texttt{bool} \mid \tau \to \tau$	
Mixin types:	$\mathcal{T} ::= \tau$	core type
	$\mid \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$	mixin signature
Type assignments:	$\mathcal{I}, \mathcal{O} ::= X_i \overset{i \in I}{\mapsto} \mathcal{T}_i$	
Dependency graphs:	$\mathcal{D}$ (see section 3.2)	

Fig. 1. Syntax of CMS_v

3.1 Syntax

The syntax of CMS_v terms and types is defined in figure 1. Here, x ranges over a countable set *Vars* of (α-convertible) variables, while X ranges over a countable set *Names* of (non-convertible) names used to identify mixin components.

Although our module system is largely independent of the core language, for the sake of specificity we use a standard simply-typed λ-calculus with constants as core language. Core terms can refer by name to a (core) component of a mixin structure, via the notation $E.X$.

Mixin terms include core terms (proper stratification of the language is enforced by the typing rules), structure expressions building a mixin from a collection of components, and the various mixin operators mentioned in section 2: sum, rename, freeze, delete and close.

A mixin structure $\langle \iota; o \rangle$ is composed of an *input assignment* ι and an *output assignment* o. The input assignment associates internal variables to names of imported components, while the output assignment associates expressions to names of exported components. These expressions can refer to imported components via their associated internal variables. This explicit distinction between names and internal variables allows internal variables to be renamed by α-conversion, while external names remain immutable, thus making projection by name unambiguous [19, 2, 21].

Due to late binding, a virtual (defined but not frozen) component of a mixin is both imported and exported by the mixin: it is exported with its current

definition, but is also imported so that other exported components refer to its final value at the time the component is frozen or the mixin is closed, rather than to its current value. In other terms, component X of $\langle \iota; o \rangle$ is deferred when $X \in cod(\iota) \setminus dom(o)$, virtual when $X \in cod(\iota) \cap dom(o)$, and frozen when $X \in dom(o) \setminus cod(\iota)$.

For example, consider the following mixin, expressed in the ML-like syntax of section 2: `mix ?val x: int let y = x + 2 let z = y + 1 end`. It is expressed in CMS_v syntax as the structure $\langle \iota; o \rangle$, where $\iota = [x \mapsto X;\ y \mapsto Y;\ z \mapsto Z]$ and $o = [Y \mapsto x + 2;\ Z \mapsto y + 1]$. The names X, Y, Z correspond to the variables in the ML-like syntax, while the variables x, y, z bind them locally. Here, X is only an input, but Y and Z are both input and output, since these components are virtual. The definition of Z refers to the imported value of Y, thus allowing later redefinition of Y to affect Z.

3.2 Types and dependency graphs

Types $\mathcal{T}$ are either core types (those of the simply-typed λ-calculus) or mixin signatures $\{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$. The latter are composed of two mappings $\mathcal{I}$ and $\mathcal{O}$ from names to types, one for input components, the other for output components; and a safe dependency graph $\mathcal{D}$.

A dependency graph $\mathcal{D}$ is a directed multi-graph whose nodes are external names of imported or exported components, and whose edges carry a valuation $\chi \in \{0, 1\}$. An edge $X \xrightarrow{1} Y$ means that the term E defining Y refers to the value of X, but in such a way that it is safe to put E in a recursive definition that simultaneously defines X in terms of Y. An edge $X \xrightarrow{0} Y$ means that the term E defining Y cannot be put in such a recursive definition: the value of X must be entirely computed before E is evaluated. It is generally undecidable whether a dependency is of the 0 or 1 kind, so we take the following conservative approximation: if E is an abstraction $\lambda x.C$, then all dependencies for Y are labeled 1; in all other cases, they are all labeled 0. (Other, more precise approximations are possible, but this one works well enough and is consistent with core ML.)

More formally, for $x \in FV(E)$, we define $\nu(x,\ E) = 1$ if $E = \lambda y.C$ and $\nu(x,\ E) = 0$ otherwise. Given the mixin structure $s = \langle \iota; o \rangle$, we then define its dependency graph $\mathcal{D}(s)$ as follows: its nodes are the names of all components of s, and it contains an edge $X \xrightarrow{\chi} Y$ if and only if there exist E and x such that $o(Y) = E$ and $\iota(x) = X$ and $x \in FV(E)$ and $\chi = \nu(x,\ E)$. We then say that a dependency graph $\mathcal{D}$ is *safe*, and write $\vdash \mathcal{D}$, if all cycles of $\mathcal{D}$ are composed of edges labeled 1. This captures the idea that only dependencies of the "1" kind are allowed inside a mutually recursive definition.

In order to type-check mixin operators, we must be able to compute the dependency graph for the result of the operator given the dependency graphs for its operands. We now define the graph-level operators corresponding to the mixin operators.

Sum: the sum $\mathcal{D}_1 + \mathcal{D}_2$ of two dependency graphs is simply their union:

$$\mathcal{D}_1 + \mathcal{D}_2 = \{X \xrightarrow{\chi} Y \mid (X \xrightarrow{\chi} Y) \in \mathcal{D}_1 \text{ or } (X \xrightarrow{\chi} Y) \in \mathcal{D}_2\}$$

Rename: assuming Y is not mentioned in $\mathcal{D}$, the graph $\mathcal{D}[X \leftarrow Y]$ is the graph $\mathcal{D}$ where the node X, if any, is renamed Y, keeping all edges unchanged.

$$\mathcal{D}[X \leftarrow Y] = \{A\{X \leftarrow Y\} \xrightarrow{\chi} B\{X \leftarrow Y\} \mid (A \xrightarrow{\chi} B) \in \mathcal{D}\}$$

Delete: the graph $\mathcal{D}\backslash X$ is the graph $\mathcal{D}$ where we remove all edges leading to X.

$$\mathcal{D}\backslash X = \mathcal{D} \setminus \{Y \xrightarrow{\chi} X \mid Y \in \mathit{Names}, \chi \in \{0,1\}\}$$

Freeze: operationally, the effect of freezing the component X in a mixin structure is to replace X by its current definition E in all definitions of other exported components. At the dependency level, this causes all components Y that previously depended on X to now depend on the names on which E depends. Thus, paths $Y \xrightarrow{\chi_1} X \xrightarrow{\chi_2} Z$ in the original graph become edges $Y \xrightarrow{\min(\chi_1,\chi_2)} Z$ in the result graph.

$$\begin{aligned} \mathcal{D}!X &= (\mathcal{D} \cup \mathcal{D}_{around}) \setminus \mathcal{D}_{remove} \\ \text{where } \mathcal{D}_{around} &= \{Y \xrightarrow{\min(\chi_1,\chi_2)} Z \mid (Y \xrightarrow{\chi_1} X) \in \mathcal{D},\ (X \xrightarrow{\chi_2} Z) \in \mathcal{D}\} \\ \text{and} \quad \mathcal{D}_{remove} &= \{X \xrightarrow{\chi} Y \mid Y \in \mathit{Names},\ \chi \in \{0,1\}\} \end{aligned}$$

The sum of two safe graphs is not necessarily safe (unsafe cycles may appear); thus, the typing rules explicitly check the safety of the sum. However, it is interesting to note that the other graph operations preserve safety:

Lemma 1 *If $\mathcal{D}$ is a safe dependency graph, then the graphs $\mathcal{D}[X \leftarrow Y]$, $\mathcal{D}\backslash X$ and $\mathcal{D}!X$ are safe.*

3.3 Typing rules

The typing rules for CMS_v are shown in figure 2. The typing environment Γ is a finite map from variables to types. We assume given a mapping TC from constants to core types. All dependency graphs appearing in the typing environment and in input signatures are assumed to be safe.

The rules resemble those of [3], with additional manipulations of dependency graphs. Projection of a structure component requires that the structure has no input components. Structure construction type-checks every output component in an environment enriched with the types assigned to the input components; it also checks that the corresponding dependency graph is safe. For the sum operator, both mixins must agree on the types of common input components, and must have no output components in common; again, we need to check that the dependency graph of the sum is safe, to make sure that the sum introduces no illegal recursive definitions. Freezing a component requires that its type in the input signature and in the output signature of the structure are identical, then removes it from the input signature. In contrast, deleting a component removes it from the output signature. Finally, closing a mixin is equivalent to freezing all its input components, and results in an empty input signature and dependency graph.

$$\Gamma \vdash x : \Gamma(x) \ \text{(var)} \qquad \Gamma \vdash c : TC(c) \ \text{(const)} \qquad \frac{\Gamma + \{x : \tau_1\} \vdash C : \tau_2}{\Gamma \vdash \lambda x.C : \tau_1 \to \tau_2} \ \text{(abstr)}$$

$$\frac{\Gamma \vdash C_1 : \tau' \to \tau \qquad \Gamma \vdash C_2 : \tau'}{\Gamma \vdash C_1 \ C_2 : \tau} \ \text{(app)} \qquad \frac{\Gamma \vdash E : \{\emptyset; \mathcal{O}; \emptyset\}}{\Gamma \vdash E.X : \mathcal{O}(X)} \ \text{(select)}$$

$$\frac{\vdash \mathcal{D}\langle \iota; o \rangle \qquad dom(o) = dom(\mathcal{O}) \qquad \Gamma + \{x : \mathcal{I}(\iota(x)) \mid x \in dom(\iota)\} \vdash o(X) : \mathcal{O}(X) \text{ for } X \in dom(o)}{\Gamma \vdash \langle \iota; o \rangle : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\langle \iota; o \rangle\}} \ \text{(struct)}$$

$$\frac{\Gamma \vdash E_1 : \{\mathcal{I}_1; \mathcal{O}_1; \mathcal{D}_1\} \qquad \Gamma \vdash E_2 : \{\mathcal{I}_2; \mathcal{O}_2; \mathcal{D}_2\} \qquad \vdash \mathcal{D}_1 + \mathcal{D}_2 \qquad dom(\mathcal{O}_1) \cap dom(\mathcal{O}_2) = \emptyset \qquad \mathcal{I}_1(X) = \mathcal{I}_2(X) \text{ for all } X \in dom(\mathcal{I}_1) \cap dom(\mathcal{I}_2)}{\Gamma \vdash E_1 + E_2 : \{\mathcal{I}_1 + \mathcal{I}_2; \mathcal{O}_1 + \mathcal{O}_2; \mathcal{D}_1 + \mathcal{D}_2\}} \ \text{(sum)}$$

$$\frac{\Gamma \vdash E : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \qquad \mathcal{I}(X) = \mathcal{O}(X)}{\Gamma \vdash E \,!\, X : \{\mathcal{I}_{\setminus X}; \mathcal{O}; \mathcal{D}!X\}} \ \text{(freeze)}$$

$$\frac{\Gamma \vdash E : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \qquad X \in dom(\mathcal{O})}{\Gamma \vdash E \setminus X : \{\mathcal{I}; \mathcal{O}_{\setminus X}; \mathcal{D} \setminus X\}} \ \text{(delete)}$$

$$\frac{\Gamma \vdash E : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \qquad Y \notin dom(\mathcal{I}) \cup dom(\mathcal{O})}{\Gamma \vdash E[X \leftarrow Y] : \{\mathcal{I} \circ [Y \mapsto X]; \mathcal{O} \circ [Y \mapsto X]; \mathcal{D}[X \leftarrow Y]\}} \ \text{(rename)}$$

$$\frac{\Gamma \vdash E : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \qquad dom(\mathcal{I}) \subseteq dom(\mathcal{O}) \qquad \mathcal{I}(X) = \mathcal{O}(X) \text{ for all } X \in dom(\mathcal{I})}{\Gamma \vdash \texttt{close}(E) : \{\emptyset; \mathcal{O}; \emptyset\}} \ \text{(close)}$$

Fig. 2. Typing rules

4 Compilation

We now present a compilation scheme translating CMS_v terms into call-by-value λ-calculus extended with records and a `let rec` binding. This compilation scheme is compositional, and type-directed, thus supporting separate compilation.

4.1 Intuitions

A mixin structure is translated into a record, with one field per output component of the structure. Each field corresponds to the expression defining the output component, but λ-abstracts all input components on which it depends, that is, all its direct predecessors in the dependency graph. These extra parameters account for the late binding semantics of virtual components. Consider again the `M1` and `M2` example at the end of section 2. These two structures are translated to:

```
m1 = { f = λg.λx. ...g...;  u = λf. f 0 }
```

```
m2 = { g = λf.λx. ...f...;  v = λg. g 1 }
```

The sum `M = M1 + M2` is then translated into a record that takes the union of the two records `m1` and `m2`:

```
m = { f = m1.f; u = m1.u; g = m2.g; v = m2.v }
```

Later, we close `M`. This requires connecting the formal parameters representing input components with the record fields corresponding to the output components. To do this, we examine the dependency graph of `M`, identifying the strongly connected components and performing a topological sort. We thus see that we must first take a fixpoint over the `f` and `g` components, then compute `u` and `v` sequentially. Thus, we obtain the following code for `close(M)`:

```
let rec f = m.f g and g = m.g f in
let u = m.u f in
let v = m.v g in
{ f = f; g = g; u = u; v = v }
```

Notice that the `let rec` definition we generate is unusual: it involves function applications in the right-hand sides, which is usually not supported in call-by-value λ-calculi. Fortunately, Boudol [6] has already developed a non-standard call-by-value calculus that supports such `let rec` definitions; we adopt a variant of his calculus as our target language.

4.2 The target language

The target language for our translation is the λ_B calculus, a variant of the λ-calculus with records and recursive definitions studied in [6]. Its syntax is as follows:

$$\begin{aligned} M ::= {} & x \mid cst \mid \lambda x.M \mid M_1\ M_2 \\ & \mid \langle X_1 = M_1; \ldots; X_n = M_n \rangle \mid M.X \\ & \mid \textbf{let}\ x = M_1\ \textbf{in}\ M \\ & \mid \textbf{let rec}\ x_1 = M_1\ \textbf{and}\ \ldots\ \textbf{and}\ x_n = M_n\ \textbf{in}\ M \end{aligned}$$

4.3 The translation

The translation scheme for our language is defined in figure 3. The translation is type-directed and operates on terms annotated by their types. For the core language constructs (variables, constants, abstractions, applications), the translation is a simple morphism; the corresponding cases are omitted from figure 3.

Access to a structure component $E.X$ is translated into an access to field X of the record obtained by translating E. Conversely, a structure $\langle \iota; o \rangle$ is translated into a record construction. The resulting record has one field for each exported name $X \in dom(o)$, and this field is associated to $o(X)$ where all input parameters on which X depends are λ-abstracted. Some notation is required here. We write $\mathcal{D}^{-1}(X)$ for the list of immediate predecessors of node X in the dependency graph $\mathcal{D}$, ordered lexicographically. (The ordering is needed

$$
\begin{array}{l}
[\![(E : \mathcal{T}').X : \mathcal{T}]\!] = [\![E : \mathcal{T}']\!].X \\
[\![\langle \iota; o\rangle : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}]\!] = \\
\quad \langle X = \boldsymbol{\lambda}\iota^{-1}(\mathcal{D}^{-1}(X)).[\![o(X) : \mathcal{O}(X)]\!] \mid X \in \mathit{dom}(\mathcal{O})\rangle \\
[\![(E_1 : \{\mathcal{I}_1; \mathcal{O}_1; \mathcal{D}_1\}) + (E_2 : \{\mathcal{I}_2; \mathcal{O}_2; \mathcal{D}_2\}) : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}]\!] = \\
\quad \textbf{let } e_1 = [\![E_1 : \{\mathcal{I}_1; \mathcal{O}_1; \mathcal{D}_1\}]\!] \textbf{ in let } e_2 = [\![E_2 : \{\mathcal{I}_2; \mathcal{O}_2; \mathcal{D}_2\}]\!] \textbf{ in} \\
\quad \langle X = e_1.X \mid X \in \mathit{dom}(\mathcal{O}_1); \\
\quad\ Y = e_2.Y \mid Y \in \mathit{dom}(\mathcal{O}_2)\rangle \\
[\![(E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}) \setminus X : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}]\!] = \\
\quad \textbf{let } e = [\![E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}]\!] \textbf{ in } \langle Y = e.Y \mid Y \in \mathit{dom}(\mathcal{O})\rangle \\
[\![(E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\})[X \leftarrow Y] : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}]\!] = \\
\quad \textbf{let } e = [\![E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}]\!] \textbf{ in} \\
\quad \langle Z\{X \leftarrow Y\} = \boldsymbol{\lambda}\overline{\mathcal{D}^{-1}(Z\{X \leftarrow Y\})}.(e.Z\ \overline{\mathcal{D}'^{-1}(Z)})\{\overline{X} \leftarrow \overline{Y}\} \mid Z \in \mathit{dom}(\mathcal{O}')\rangle \\
[\![(E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\})\,!\,X : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}]\!] = \\
\quad \textbf{let } e = [\![E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}]\!] \textbf{ in} \\
\quad \langle Z = e.Z \mid Z \in \mathit{dom}(\mathcal{O}),\ X \notin \mathcal{D}'^{-1}(Z); \\
\quad\ Y = \boldsymbol{\lambda}\overline{\mathcal{D}^{-1}(Y)}.\textbf{let rec } \overline{X} = e.X\ \overline{\mathcal{D}'^{-1}(X)} \textbf{ in } e.Y\ \overline{\mathcal{D}'^{-1}(Y)} \mid X \in \mathcal{D}'^{-1}(Y)\rangle \\
[\![\texttt{close}(E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}) : \{\emptyset; \mathcal{O}; \emptyset\}]\!] = \\
\quad \textbf{let } e = [\![E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}]\!] \textbf{ in} \\
\quad \textbf{let rec } \overline{X_1^1} = e.X_1^1\ \overline{\mathcal{D}'^{-1}(X_1^1)} \textbf{ and } \ldots \textbf{ and } \overline{X_{n_1}^1} = e.X_{n_1}^1\ \overline{\mathcal{D}'^{-1}(X_{n_1}^1)} \textbf{ in} \\
\quad \ldots \\
\quad \textbf{let rec } \overline{X_1^p} = e.X_1^p\ \overline{\mathcal{D}'^{-1}(X_1^p)} \textbf{ and } \ldots \textbf{ and } \overline{X_{n_p}^p} = e.X_{n_p}^p\ \overline{\mathcal{D}'^{-1}(X_{n_p}^p)} \textbf{ in} \\
\quad \langle X = \overline{X} \mid X \in \mathit{dom}(\mathcal{O})\rangle \\
\quad \text{where } (\{X_1^1 \ldots X_{n_1}^1\}, \ldots, \{X_1^p \ldots X_{n_p}^p\}) \text{ is a serialization of } \mathit{dom}(\mathcal{O}') \text{ against } \mathcal{D}'
\end{array}
$$

Fig. 3. The translation scheme

to ensure that values for these predecessors are provided in the correct order later; any fixed total ordering will do.) If $(X_1, \ldots, X_n) = \mathcal{D}^{-1}(X)$ is such a list, we write $\iota^{-1}(\mathcal{D}^{-1}(X))$ for the list $(x_1, \ldots, x_n)$ of variables associated to the names $(X_1, \ldots, X_n)$ by the input mapping ι. Finally, we write $\boldsymbol{\lambda}(x_1, \ldots, x_n).M$ as shorthand for $\lambda x_1 \ldots \lambda x_n.M$. With all this notation, the field X in the record translating $\langle \iota; o\rangle$ is bound to $\boldsymbol{\lambda}\iota^{-1}(\mathcal{D}^{-1}(X)).[\![o(X) : \mathcal{O}(X)]\!]$.

The sum of two mixins $E_1 + E_2$ is translated by building a record containing the union of the fields of the translations of E_1 and E_2. For the delete operator $E \setminus X$, we return a copy of the record representing E in which the field X is omitted. Renaming $E[X \leftarrow Y]$ is harder: not only do we need to rename the field X of the record representing E into Y, but the renaming of X to Y in the input parameters can cause the order of the implicit arguments of the record fields to change. Thus, we need to abstract again over these parameters in the correct

order after the renaming, then apply the corresponding field of $[\![E]\!]$ to these parameters in the correct order before the renaming. Again, some notation is in order: to each name X we associate a fresh variable written $\overline{X}$, and similarly for lists of names, which become lists of variables. Moreover, we write $M\ (x_1, \ldots, x_n)$ as shorthand for $M\ x_1\ \ldots\ x_n$.

The freeze operation $E\,!\,X$ is perhaps the hardest to compile. Output components Z that do not depend on X are simply re-exported from $[\![E]\!]$. For the other output components, consider a component Y of E that depends on $Y_1, \ldots, Y_n$, and assume that one of these dependencies is X, which itself depends on $X_1, \ldots, X_p$. In $E\,!\,X$, the Y component depends on $(\{Y_i\} \cup \{X_j\}) \setminus \{X\}$. Thus, we λ-abstract on the corresponding variables, then compute X by applying $[\![E]\!].X$ to the parameters $\overline{X_j}$. Since X can depend on itself, this application must be done in a **let rec** binding over $\overline{X}$. Then, we apply $[\![E]\!].Y$ to the parameters that it expects, namely $\overline{Y_i}$, which include $\overline{X}$.

The only operator that remains to be explained is `close`(E). Here, we take advantage of the fact that `close` removes all input dependencies to generate code that is more efficient than a sequence of freeze operations. We first *serialize* the set of names exported by E against its dependency graph $\mathcal{D}$. That is, we identify strongly connected components of $\mathcal{D}$, then sort them in topological order. The result is an enumeration $(\{X_1^1 \ldots X_{n_1}^1\}, \ldots, \{X_1^p \ldots X_{n_p}^p\})$ of the exported names where each cluster $\{X_1^i \ldots X_{n_i}^i\}$ represents mutually recursive definitions, and the clusters are listed in an order such that each cluster depends only on the preceding ones. We then generate a sequence of **let rec** bindings, one for each cluster, in the order above. In the end, all output components are bound to values with no dependencies, and can be grouped together in a record.

5 Type soundness of the translation

The translation scheme defined above can generate recursive definitions of the form **let rec** $x = M\ x$ **in** N. In λ_B, these definitions can either evaluate to a fixpoint (*i.e.* $M = \lambda x.\lambda y.y$), or get stuck (*i.e.* $M = \lambda x.x + 1$). In the full paper, we prove that no term generated by the translation of a well-typed mixin can get stuck. To this end, we equip λ_B with a sound type system that guarantees that all recursive definitions are correct. Boudol [6] gave such a type system, using function types of the form $\tau_1 \xrightarrow{0} \tau_2$ or $\tau_1 \xrightarrow{1} \tau_2$ to denote functions that respectively do or do not inspect the value of their argument immediately when applied. However, this type system does not type-check curried function applications with sufficient precision for our purposes. Therefore, we developed a refinement of this type system based on function types of the form $\tau_1 \xrightarrow{n} \tau_2$, where n is an integer indicating the number of applications that can be performed without inspecting the value of the first argument. In the full paper [17], we formally define this type system, show its soundness (well-typed terms do not get stuck), and prove that λ_B terms produced by the compilation scheme applied to well-typed mixins are well typed.

6 Related work

Bracha [8, 7] introduced the concept of mixin as a generalization of (multiple) inheritance in class-based OO languages, allowing more freedom in deferring the definition of a method in a class and implementing it later in another class than is normally possible with inheritance and overriding.

Duggan and Sourelis [13, 14] were the first to transpose Bracha's mixin concept to the ML module system. Their mixin module system supports extensible functions and datatypes: a function defined by cases can be split across several mixins, each mixin defining only certain cases, and similarly a datatype (sum type) can be split across several mixins, each mixin defining only certain constructors; a composition operator then stitches together these cases and constructors. The problem with ill-founded recursions is avoided by allowing only functions (λ-abstractions) in the combinable parts of mixins, while initialization code goes into a separate, non-combinable part of mixins. Their compilation scheme (into ML modules) is less efficient than ours, since the fixpoint defining a function is computed at each call, rather than only once at mixin combination time as in our system.

The *units* of Flatt and Felleisen [15] are a module system for Scheme. The basic program units import names and export definitions, much like in Ancona and Zucca's *CMS* calculus. The recursion problem is solved as in [13] by separating initialization from component definition.

Ancona and Zucca [1–3] develop a theory of mixins, abstracting over much of the core language, and show that it can encode the pure λ-calculus, as well as Abadi and Cardelli's object calculus. The emphasis is on providing a calculus, with reduction rules but no fixed reduction strategy, and nice confluence properties. Another calculus of mixins is Vestergaard and Wells' `m`-calculus [21], which is very similar to *CMS* in many points, but is not based on any core language, using only variables instead. The emphasis is put on the equational theory, allowing for example to replace some variables with their definition inside a structure, or to garbage collect unused components, yielding a powerful theory. Neither Ancona-Zucca nor Vestergaard-Wells attempt to control recursive definitions statically, performing on-demand unwinding instead. Still, some care is required when unwinding definitions inside a structure, because of confluence problems [4].

Crary *et al* [11, 12] and Russo [20] extend the Standard ML module system with mutually recursive structures via a `structure rec` binding. Like mixins, this construct addresses ML's cross-module recursion problem; unlike mixins, it does not support late binding and incremental programming. The `structure rec` binding does not lend itself directly to separate compilation (the definitions of all mutually recursive modules must reside in the same source file), although some amount of separate compilation can be achieved by functorizing each recursive module over the others. ML structures contain type components in addition to value components, and this raises delicate static typing issues that we have not yet addressed within our CMS_v framework. Crary *et al* formalize static typing of recursive structure using recursively-defined signatures and the phase

distinction calculus, while Russo remains closer to Standard ML's static semantics. Concerning ill-founded recursive value definitions, Russo does not attempt to detect them statically, relying on lazy evaluation to catch them at run-time. Crary *et al* statically require that all components of recursive structures are syntactic values. This is safe, but less flexible than our component-per-component dependency analysis.

Bono *et al* [5] use a notion of dependency graph in the context of a type system for extensible and incomplete objects. However, they do not distinguish between "0" and "1" dependencies.

7 Conclusions and future work

As a first step towards a full mixin module system for ML, we have developed a call-by-value variant of Ancona and Zucca's calculus of mixins. The main technical innovation of our work is the use of dependency graphs in mixin signatures, statically guaranteeing that cross-module recursive definitions are well founded, yet leaving maximal flexibility in mixing recursive function definitions and non-recursive computations within a single mixin. Dependency graphs also allow a separate compilation scheme for mixins where fixpoints are taken as early as possible, *i.e.* during mixin initialization rather than at each component access.

A drawback of dependency graphs is that programmers must (in principle) provide them explicitly when declaring a mixin signature, *e.g.* for a deferred sub-mixin component. This could make programs quite verbose. Future work includes the design of a concrete syntax for mixin signatures that alleviate this problem in the most common cases.

Our λ_B target calculus can be compiled efficiently down to machine code, using the "in-place updating" trick described in [10] to implement the non-standard `let rec` construct. However, this trick assumes constant-sized function closures; some work is needed to accommodate variable-sized closures as used in the OCaml compiler among others.

The next step towards mixin modules for ML is to support type definitions and declarations as components of mixins. While these type components account for most of the complexity of ML module typing, we are confident that we can extend to mixins the considerable body of type-theoretic work already done for ML modules [16, 18] and recursive modules [11, 12].

Acknowledgements. We thank Elena Zucca and Davide Ancona for discussions, and Vincent Simonet for his technical advice on the typing rules for λ_B.

References

1. D. Ancona. *Modular formal frameworks for module systems*. PhD thesis, Universita di Pisa, 1998.
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. *Journal of functional programming*, 2001. To appear.
4. Z. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 2001. To appear.
5. V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping for extensible, incomplete objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.
6. G. Boudol. The recursive record semantics of objects revisited. Research report 4199, INRIA, 2001. Preliminary version presented at ESOP'01, LNCS 2028.
7. G. Bracha. *The programming language Jigsaw: mixins, modularity and multiple inheritance*. PhD thesis, University of Utah, 1992.
8. G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA90*, volume 25(10) of *SIGPLAN Notices*, pages 303–311. ACM Press, 1990.
9. L. Cardelli. Program fragments, linking, and modularization. In *24th symp. Principles of Progr. Lang*, pages 266–277. ACM Press, 1997.
10. G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.
11. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Prog. Lang. Design and Impl. 1999*, pages 50–63. ACM Press, 1999.
12. D. Dreyer, K. Crary, and R. Harper. Toward a practical type theory for recursive modules. Technical Report CMU-CS-01-112, Carnegie Mellon University, 2001.
13. D. Duggan and C. Sourelis. Mixin modules. In *Int. Conf. on Functional Progr. 96*, pages 262–273. ACM Press, 1996.
14. D. Duggan and C. Sourelis. Recursive modules and mixin-based inheritance. Unpublished draft, 2001.
15. M. Flatt and M. Felleisen. Units: cool modules for HOT languages. In *Prog. Lang. Design and Impl. 1998*, pages 236–248. ACM Press, 1998.
16. 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.
17. T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting (long version). Available at `http://pauillac.inria.fr/~hirschow`, 2001.
18. X. Leroy. Manifest types, modules, and separate compilation. In *21st symp. Principles of Progr. Lang*, pages 109–122. ACM Press, 1994.
19. M. Lillibridge. *Translucent sums : a foundation for higher-order module systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.
20. C. Russo. Recursive structures for Standard ML. In *Int. Conf. on Functional Progr. 01*, pages 50–61, 2001.
21. J. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Programming Languages and Systems, 9th European Symp. Programming*, volume 1782 of *LNCS*, pages 412–428. Springer-Verlag, 2000.