Mixin Modules and Computational Effects

D.Ancona, S.Fagorzi, E.Moggi, E.Zucca*

DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy
email: {davide,fagorzi,moggi,zucca}@disi.unige.it

Abstract. We define a calculus for investigating the interactions between mixin modules and computational effects, by combining the purely functional mixin calculus *CMS* with a monadic metalanguage supporting the two separate notions of *simplification* (local rewrite rules) and *computation* (global evaluation able to modify the store). This distinction is important for smoothly integrating the *CMS* rules (which are all local) with the rules dealing with the imperative features.
In our calculus mixins can contain mutually recursive *computational* components which are explicitly computed by means of a new mixin operator whose semantics is defined in terms of a Haskell-like recursive monadic binding. Since we mainly focus on the operational aspects, we adopt a simple type system like that for Haskell, that does not detect dynamic errors related to bad recursive declarations involving effects. The calculus serves as a formal basis for defining the semantics of imperative programming languages supporting first class mixins while preserving the *CMS* equational reasoning.

1 Introduction

Mixin modules (or simply mixins) are modules supporting *parameterization*, *cross-module recursion* and *overriding* with *late binding*; these three features altogether make mixin module systems a valuable tool for promoting software reuse and incremental programming [AZ02]. As a consequence, there have been several proposals for extending existing languages with mixins; however, even though there already exist some prototype implementations of such extensions (see, e.g., [FF98a,FF98b,HL02]), there are still several problems to be solved in order to fully and smoothly integrate mixins with all the other features of a real language. For instance, in the presence of store manipulation primitives, expressions inside mixins can have side-effects, but this possibility raises some semantic issues: (1) because of side-effects, the *evaluation* order of components inside a mixin must be *deterministic*, while still retaining cross-module-recursion; (2) *when* computations inside a mixin must be evaluated and *how many times*?
Unfortunately, all formalizations defined so far [AZ99,AZ02,MT00,WV00] do not consider these issues, since they only model mixins in purely functional settings.
In this paper we propose a monadic mixin calculus, called $CMS_{\mathtt{do}}$, for studying the interaction between the notions of mixin and store. More precisely, this calculus should serve as a formal basis both for defining the semantics of imperative programming languages supporting mixins and for allowing equational reasoning.
Our approach consists in combining the purely functional mixin calculus *CMS* [AZ99,AZ02] with a monadic metalanguage [MF03] equipped with a Haskell-like recursive monadic binding [EL00,EL02] and supporting the two separate notions of *simplification* and *computation*, the former corresponding to local rewriting with no side-effects, the latter to global evaluation steps able to modify the store. This distinction is important for smoothly integrating the *CMS* rules (which are all

* Supported by MIUR project NAPOLI, EU project DART IST-2001-33477 and thematic network APPSEM II IST-2001-38957

local) with the rules dealing with the imperative features; furthermore, since simplification is a congruence, all *CMS* equations (except those related to selection) hold in CMS_{do}.
In CMS_{do} a mixin can contain, besides the usual *CMS* definitions, also *computational* definitions of the form $x \Leftarrow e$, where e has monadic type. The (simplification) rules for the standard operators on mixins coincide with those given for *CMS*. However, before selecting components from a mixin, this must be transformed into a record. The transformation of a mixin (without deferred components) into a record is triggered by the `doall` primitive, and consists in (1) evaluating computational definitions $x_i \Leftarrow e_i$ in the order they are declared; (2) binding the value returned by e_i to x_i immediately, to make it available to the subsequent computations e_j with $j > i$.
Mutual recursion has the following informal semantics: if $i \leq j$, then e_i can depend on the variable x_j, provided that the computation e_i can be successfully performed without knowing the value of e_j (which is bound to x_j only later). Formally, the semantics of `doall` is expressed in terms of a recursive monadic binding, similar to that defined in [EL00,EL02], and a standard recursive let-binding.
Since the emphasis of the paper is on the operational aspects, we adopt a simple type system like that for Haskell, that does not detect dynamic errors related to bad recursive declarations; for instance, $\mathsf{doall}([; x \Leftarrow \mathsf{set}(y, 1), y \Leftarrow \mathsf{new}(0)])$ is a well-typed term which evaluates into a dynamic error. However, more refined type systems based on dependencies analysis [Bou02,HL02] could be considered for CMS_{do} in order to avoid this kind of dynamic errors.
The rest of the paper is organized as follows. In Section 2 we illustrate the main features of the original *CMS* calculus and introduce the new CMS_{do} calculus through some examples. In Section 3 we formally define the syntax of the calculus, the type system and the two relations of simplification and computation. We also prove standard technical results, including a bisimulation result (simplification does not affect computation steps) and the progress property for the combined relation. In Section 4 we discuss related work and in Section 5 we summarize the contribution of the paper and draw some further research directions.

2 An Overview of the Calculus

In this section we give an overview of the CMS_{do} calculus by means of some examples written in a more user-friendly syntax.
Like in *CMS*, a CMS_{do} *basic mixin module* consists of *defined* and *local* components, bound to an expression, and *deferred* components, declared but not yet defined.

Example 1. For instance,

```
M1 = mix import N2 as x,              (* deferred *)
         export N1 = e1[x,y],         (* defined *)
         local   y = e2[x,y]         (* local *)
     end
```

denotes a mixin with one deferred, one defined and one local[1] component, where `e1[x,y]` and `e2[x,y]` denote two arbitrary expressions possibly containing the two free variables `x` and `y`. Deferred components are associated with both a component name (as `N2`) and a variable (as `x`); component names are used for external referencing of deferred and defined components but they are not expressions, while variables are used for accessing deferred and local components inside mixins (for further details on the separation between variables and component names see [Ler94], [HL94], [AZ02]). Local components are not visible from the outside and can be mutually recursive.

[1] Note that deferred, defined and local components can be declared in any order; in particular, definitions of defined and local components can be interleaved.

Besides this construct, $CMS_{\mathtt{do}}$ provides four operations on mixins: *sum*, *freeze*, *delete* (inherited from *CMS*) and *doall*.

Example 2. Two mixins can be combined by the *sum* operation, which performs the union of the deferred components (in the sense that components with the same name are shared), and the disjoint union of the defined and local components of the two mixins. However, while defined components must be disjoint because clashes are not allowed by the type system, the disjoint union of local components can be always performed by renaming variables.

```
M2 = mix import N1 as x,
         export N2 = e3[x,y],
         local   y = e4[x,y]
     end
M3 = M1 + M2
```

Module `M3` simplifies to

```
mix import N2 as x1, N1 as x2,
    export N1 = e1[x1,y1],
    local  y1 = e2[x1,y1],
    export N2 = e3[x2,y2],
    local  y2 = e4[x2,y2]
end
```

The sum operation supports cross-module recursion; in module `M3`, the definition of `N2`, which is needed by `M1`, is provided by `M2`, whereas the definition of `N1`, which is needed by `M2`, is provided by `M1`. However, in $CMS_{\mathtt{do}}$ component selection is permitted only if the module has no deferred components, therefore the defined components of `M3` cannot be selected even though the deferred components of `M3` (`N1` and `N2`) are also among the defined ones.

Example 3. The *freeze* operation connects deferred and defined components having the same name inside a mixin; in other words, it is used for resolving "external names", so that a deferred component becomes local.
For instance, in `(mix import N as x, export N = e1[x,y], local y = e2[x,y]) ! N` the deferred component `N` has been effectively bound to the corresponding defined component by freezing it, obtaining the following simplified expression:

```
mix local x = e1[x,y], export N = x, local y = e2[x,y] end
```

Example 4. The *delete* operation is used for hiding defined components:

```
(mix import N as x, export N = e1[x,y], local y = e2[x,y]) \ N
```

simplifies to

```
mix import N as x, local  y = e2[x,y] end
```

So far the calculus is very similar to the pure functional calculus *CMS* defined in [AZ02]; its primitive operations can be used for expressing a variety of convenient constructs supporting cross-module recursion and overriding with late binding.
For instance, `M6 = (((M3 \ N2) + mix export N2 = e[] end) ! N1) ! N2` corresponds to declare a new mixin obtained from `M3` by overriding component `N2`; since `N2` in `M3` is both deferred and defined, the definition of component `N2` in `M6` depends on the new definition of `N2` in `M6` rather than on that in `M3` (late binding). We refer to [AZ02] for more details on this.
In addition to the *CMS* operations and constructs presented above, $CMS_{\mathtt{do}}$ provides a new kind of mixin component called *computational*, a new mixin operation *doall* to deal with computational components, the usual primitives on the store, and the monadic constructs `mdo` (recursive *do*) and `ret` (embedding of values into computations).

Example 5. Let us consider the following mixin definition:

```
CM1 = mix local lc <= new(x-1), x = 1,
          export Inc = mdo v <= get(lc) in set(lc,v+1), Val <= get(lc)
      end
```

The local component `lc` and the defined component `Val` has been defined via `<=` (rather than `=`) and are called *computational*.
Evaluation of computational components like `lc` and `Val` can be performed only once by means of the *doall* operation (see below), provided that there are no deferred components (as in this case); furthermore, selection of the defined components of `CM1` is possible only after `lc` and `Val` have been evaluated.
Note that `Inc` is not computational, even though its associated expression contains effects, therefore the *doall* operation does not compute `Inc` (see below).
The computation `new(x-1)` returns a fresh location containing the expression `x-1`, `get(lc)` returns the expression stored at the location l denoted by `lc` and `set(lc,v+1)` updates the store by assigning `v+1` to l and returns l. Note that `new(e)` and `set(lc,e)` are "lazy", in the sense that they do not evaluate the expression `e`.
Let us now consider the expression `doall(CM1)`; its evaluation returns a record containing only the defined components `Inc` and `Val`. As already explained, `Inc` is not evaluated, whereas `Val` is computed as follows. Since we require the evaluation of computational components to respect the declaration order, the expression associated with `lc` is computed before that defining `Val`; once the value of variable `lc` is computed it is made immediately available to the next computational component `Val`.
On the other hand, the component `Inc` (defined via `=`) is not computed, but its associated computation is treated as a value of monadic type that can be evaluated with the `mdo` construct. Therefore, if l is the location generated by the evaluation of component `lc`, then `doall(CM1)` evaluates to the record `r={Inc=mdo v<=get(`l`) in set(`l`,v+1), Val=0}`, where component `Inc` can be reevaluated several times, for instance, in the expression `mdo lc<=r.Inc in get(lc)` which increments the contents of l and evaluates to `1`. Finally, note that the order of computational components matters, while that of non-computational components, like `x` and `Inc` in `CM1`, does not.

Example 6. Computational components can be mutually recursive like in the following mixin.

```
CM2 = mix export Loc1=l1, Loc2=l2,
          local l1<=new(l2), l2<=new(l1)
      end
```

The expression `doall(CM2)` evaluates to the record `{Loc1=`l_1`, Loc2=`l_2`}` where l_1 and l_2 are two locations pointing two each other. This is possible because `new`(e) does not need to evaluate e.
On the other hand, evaluation of `doall(mix local x<=set(y,1), y<=new(0) end)` causes an error because of bad recursive declarations. In this case the error could be avoided by swapping `x` and `y`, but reordering computational components changes the semantics.

3 CMS_{do}: a monadic mixin language

Before defining CMS_{do}, we introduce some notations and conventions.

- If s_1 and s_2 are two finite sequences, then s_1, s_2 denotes their concatenation.
- $f\colon A \xrightarrow{fin} B$ means that f is a partial function from A to B with a finite domain, written $\mathsf{dom}(f)$. We write $\{a_i\colon b_i | i \in I\}$ for the partial function mapping for all $i \in I$ a_i to b_i (where the a_i must be different, i.e. $a_i = a_j$ implies $i = j$). We use the following operations on partial functions:

- $\emptyset$ is the everywhere undefined partial function;
- f and g are *compatible* when $f(x) = g(x)$ when $x \in \mathsf{dom}(f) \cap \mathsf{dom}(g)$.
- f_1, f_2 denotes the union of two compatible partial functions;
- $f\{a\colon b\}$ denotes the update of f in a;
- $f \setminus a$ is the partial function g such that $g(x) \stackrel{\Delta}{=} \begin{cases} f(x) & \text{if } x \neq a \\ \text{undefined} & \text{otherwise} \end{cases}$

– $\stackrel{*}{\longrightarrow}$ denotes the reflexive and transitive closure of a binary relation $\longrightarrow$.
– If E is a set of terms, then $\mathsf{FV}(e)$ is the set of free variables of e; E_0 is the set of $e \in \mathsf{E}$ s.t. $\mathsf{FV}(e) = \emptyset$; $e\{\rho\}$, with ρ a finite partial function from a set of variables Var to E, denotes the parallel substitution of all variables $x \in \mathsf{dom}(\rho)$ with $\rho(x)$ in e (modulo α-conversion).

The syntax of CMS_{do} definition is parametric in an infinite set Name of *component names* X (for records and mixins), an infinite set Var of variables x, and an infinite set L of *locations* l.
Terms e, recursive monadic bindings Θ and mixin bindings Δ are given by

$$\begin{array}{l} e \in \mathsf{E} ::= x \mid \{o\} \mid e.X \mid \mathsf{let}(\rho; e) \mid \mathsf{ret}(e) \mid \mathsf{mdo}\,(\Theta; e) \mid \mathsf{doall}(e) \\ \qquad \mid\ l \mid \mathsf{new}(e) \mid \mathsf{get}(e) \mid \mathsf{set}(e_1, e_2) \mid e_1 + e_2 \mid e!X \mid e \setminus X \\ \qquad \mid\ [\iota;\ \Delta] \qquad \text{with } \iota \text{ injective and } \mathsf{dom}(\iota) \cap \mathsf{DV}(\Delta) = \emptyset \\ \quad \Theta ::= \emptyset \mid \Theta, x \Leftarrow e \qquad \text{with } x \notin \mathsf{DV}(\Theta) \\ \quad \Delta ::= \emptyset \mid \Delta, D \qquad \text{with } \mathsf{DV}(\Delta) \cap \mathsf{DV}(D) = \mathsf{DN}(\Delta) \cap \mathsf{DN}(D) = \emptyset \\ \quad D ::= X \lhd e \mid x \lhd e \qquad \text{with } \lhd \text{ either } = \text{ or } \Leftarrow \end{array}$$

where $o\colon \mathsf{Name} \stackrel{fin}{\to} \mathsf{E}$, $\rho\colon \mathsf{Var} \stackrel{fin}{\to} \mathsf{E}$ and $\iota\colon \mathsf{Var} \stackrel{fin}{\to} \mathsf{Name}$. Some productions have side-conditions, the auxiliary functions DV and DN return the set of variables and component names defined in a sequence Δ of definitions, respectively. For lack of space, the straightforward definitions of DV, DN and FV have been omitted (see the long version of this paper[2]). The terms include:

– records $\{o\}$, where o is a partial function (since the order of record components is irrelevant), and selection $e.X$ of a record component;
– recursive bindings $\mathsf{let}(\rho; e)$ and recursive monadic bindings $\mathsf{mdo}\,(\Theta; e)$ of [EL00];
– the operations on references for allocation $\mathsf{new}(e)$, dereferencing $\mathsf{get}(e)$ and assignment $\mathsf{set}(e_1, e_2)$;
– basic mixins $[\iota;\ \Delta]$ with deferred components ι, and the operations of sum $e_1 + e_2$, freezing $e!X$ and deletion $e \setminus X$ of a component (see [AZ02]).

The basic difference between a record $\{o\}$ and a mixin $[\emptyset;\ \Delta]$ without deferred components is that Δ may have local (recursive) definitions and computational components. The operation $\mathsf{doall}([\emptyset;\ \Delta])$ denotes a computation which forces evaluation of all computational components in Δ (eliminates local definitions), and returns a record. Since computations may have side-effects, the order of the bindings in Δ (and Θ) matters.
Types are defined by $\boxed{\tau \in \mathsf{T} ::= \ldots \mid M\tau \mid \mathsf{ref}\tau \mid \{\Pi\} \mid [\Pi;\ \Pi']}$ where $\Pi\colon \mathsf{Name} \stackrel{fin}{\to} \mathsf{T}$. The set of types includes computational types $M\tau$, reference types, record types $\{\Pi\}$ and mixin types $[\Pi;\ \Pi']$. Table 1 gives the typing rules for deriving judgments of the form $\Gamma \vdash_\Sigma e\colon \tau$, which mean "$e$ is a well-typed term of type τ in Γ and Σ", where $\Gamma\colon \mathsf{Var} \stackrel{fin}{\to} \mathsf{T}$ is a type assignment, and $\Sigma\colon \mathsf{L} \stackrel{fin}{\to} \mathsf{T}$ is a signature for locations. The type system enjoys the usual properties of weakening (w.r.t. Γ and Σ) and substitution.

[2] `http://www.disi.unige.it/person/AnconaD/Conferences`

$$\text{(var)}\ \frac{}{\Gamma \vdash_\Sigma x : \tau}\ \Gamma(x) = \tau \qquad \text{(ret)}\ \frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathsf{ret}(e) : M\tau}$$

$$\text{(mdo)}\ \frac{\begin{array}{l}\{\Gamma, \Gamma_\Theta \vdash_\Sigma e : M\tau \mid (x \Leftarrow e) \in \Theta \wedge \tau = \Gamma_\Theta(x)\} \\ \Gamma, \Gamma_\Theta \vdash_\Sigma e' : M\tau'\end{array}}{\Gamma \vdash_\Sigma \mathsf{mdo}\,(\Theta; e') : M\tau'}\ \mathsf{dom}(\Gamma_\Theta) = \mathsf{DV}(\Theta)$$

$$\text{(let)}\ \frac{\begin{array}{l}\{\Gamma, \Gamma_\rho \vdash_\Sigma e : \tau \mid e = \rho(x) \wedge \tau = \Gamma_\rho(x)\} \\ \Gamma, \Gamma_\rho \vdash_\Sigma e' : \tau\end{array}}{\Gamma \vdash_\Sigma \mathsf{let}(\rho; e') : \tau}\ \mathsf{dom}(\Gamma_\rho) = \mathsf{dom}(\rho)$$

$$\text{(l)}\ \frac{}{\Gamma \vdash_\Sigma l : \mathsf{ref}\tau}\ \Sigma(l) = \tau \qquad \text{(new)}\ \frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathsf{new}(e) : M(\mathsf{ref}\tau)}$$

$$\text{(get)}\ \frac{\Gamma \vdash_\Sigma e : \mathsf{ref}\tau}{\Gamma \vdash_\Sigma \mathsf{get}(e) : M\tau} \qquad \text{(set)}\ \frac{\Gamma \vdash_\Sigma e_2 : \tau \quad \Gamma \vdash_\Sigma e_1 : \mathsf{ref}\tau}{\Gamma \vdash_\Sigma \mathsf{set}(e_1, e_2) : M(\mathsf{ref}\tau)}$$

$$\text{(record)}\ \frac{\{\Gamma \vdash_\Sigma e : \tau \mid e = o(X) \wedge \tau = \Pi(X)\}}{\Gamma \vdash_\Sigma \{o\} : \{\Pi\}}\ \mathsf{dom}(\Pi) = \mathsf{dom}(o)$$

$$\text{(select)}\ \frac{\Gamma \vdash_\Sigma e : \{\Pi\}}{\Gamma \vdash_\Sigma e.X : \tau}\ \tau = \Pi(X) \qquad \text{(doall)}\ \frac{\Gamma \vdash_\Sigma e : [\emptyset;\ \Pi]}{\Gamma \vdash_\Sigma \mathsf{doall}(e) : M\{\Pi\}}$$

$$\text{(mixin)}\ \frac{\begin{array}{l}\{\Gamma, \Gamma_1, \Gamma_2 \vdash_\Sigma e : \tau \mid (X = e) \in \Delta \wedge \tau = \Pi'(X)\} \\ \{\Gamma, \Gamma_1, \Gamma_2 \vdash_\Sigma e : M\tau \mid (X \Leftarrow e) \in \Delta \wedge \tau = \Pi'(X)\} \\ \{\Gamma, \Gamma_1, \Gamma_2 \vdash_\Sigma e : \tau \mid (x = e) \in \Delta \wedge \tau = \Gamma_2(x)\} \\ \{\Gamma, \Gamma_1, \Gamma_2 \vdash_\Sigma e : M\tau \mid (x \Leftarrow e) \in \Delta \wedge \tau = \Gamma_2(x)\}\end{array}}{\Gamma \vdash_\Sigma [\iota;\ \Delta] : [\Pi;\ \Pi']}\ \begin{array}{l}\mathsf{img}(\iota) = \mathsf{dom}(\Pi) \\ \Gamma_1 \triangleq \Pi \circ \iota \\ \mathsf{DN}(\Delta) = \mathsf{dom}(\Pi') \\ \mathsf{DV}(\Delta) = \mathsf{dom}(\Gamma_2)\end{array}$$

$$\text{(sum)}\ \frac{\Gamma \vdash_\Sigma e_1 : [\Pi_1;\ \Pi_1'] \quad \Gamma \vdash_\Sigma e_2 : [\Pi_2;\ \Pi_2']}{\Gamma \vdash_\Sigma e_1 + e_2 : [\Pi_1, \Pi_2;\ \Pi_1', \Pi_2']}\ \begin{array}{l}\Pi_1 \text{ compatible with } \Pi_2 \\ \mathsf{dom}(\Pi_1') \cap \mathsf{dom}(\Pi_2') = \emptyset\end{array}$$

$$\text{(freeze)}\ \frac{\Gamma \vdash_\Sigma e : [\Pi;\ \Pi']}{\Gamma \vdash_\Sigma e!X : [\Pi \setminus X;\ \Pi']}\ \Pi(X) = \Pi'(X)$$

$$\text{(delete)}\ \frac{\Gamma \vdash_\Sigma e : [\Pi;\ \Pi']}{\Gamma \vdash_\Sigma e \setminus X : [\Pi;\ \Pi' \setminus X]}\ X \in \mathsf{dom}(\Pi')$$

Table 1. Type system

3.1 Simplification

We define a confluent relation on terms (and other syntactic categories), called *simplification*, which induces a congruence on terms. There is no need to define a deterministic simplification strategy, since computational effects (in our case they amount to store changes) are *insensitive* to further simplification (see Theorem 1). Simplification $e_1 \longrightarrow e_2$ is the compatible relation on E induced by the rewrite rules in Table 2.

(R) $\{o\}.X \longrightarrow e$ provided $e = o(X)$
(L) $\mathsf{let}(\rho; e) \longrightarrow e\{x{:}\,\mathsf{let}(\rho; \rho(x)) | x \in \mathsf{dom}(\rho)\}$
(S) $[\iota_1;\, \Delta_1] + [\iota_2;\, \Delta_2] \longrightarrow [\iota_1, \iota_2;\, \Delta_1, \Delta_2]$ provided $[\iota_1, \iota_2;\, \Delta_1, \Delta_2]$ is well-formed, i.e.
- $\mathsf{DN}(\Delta_1) \cap \mathsf{DN}(\Delta_2) = \mathsf{DV}(\Delta_1) \cap \mathsf{DV}(\Delta_2) = \mathsf{dom}(\iota_1, \iota_2) \cap \mathsf{DV}(\Delta_1, \Delta_2) = \emptyset$
- ι_1, ι_2 is an injection (therefore ι_1 is compatible with ι_2)
- $\mathsf{FV}(\Delta_1) \cap (\mathsf{dom}(\iota_2) \cup \mathsf{DV}(\Delta_2)) = \mathsf{FV}(\Delta_2) \cap (\mathsf{dom}(\iota_1) \cup \mathsf{DV}(\Delta_1)) = \emptyset$

(F) $[\iota, x{:}\,X;\, \Delta, X \lhd e, \Delta']!X \longrightarrow [\iota;\, \Delta, x \lhd e, X = x, \Delta']$
(D) $[\iota;\, \Delta, X \lhd e, \Delta'] \setminus X \longrightarrow [\iota;\, \Delta, \Delta']$
(A) $\mathsf{doall}([\emptyset;\, \Delta]) \longrightarrow \mathsf{mdo}\,(|\Delta|; \mathsf{ret}\{o_1, o_2\})\{x{:}\,\mathsf{let}(\rho; x) | x \in \mathsf{dom}(\rho)\}$ where
- $\rho = \{x{:}\,e | (x = e) \in \Delta\}$
- $o_1 = \{X{:}\,e | (X = e) \in \Delta\}$, $o_2 = \{X{:}\,x_X | X \Leftarrow e \in \Delta\}$ with x_X freshly chosen
- $|\Delta|$ is defined by induction on Δ as follows:
  - $|\emptyset| = \emptyset$
  - $|(\Delta, X = e)| = |(\Delta, x = e)| = |\Delta|$
  - $|(\Delta, X \Leftarrow e)| = |\Delta|, x_X \Leftarrow e$
  - $|(\Delta, x \Leftarrow e)| = |\Delta|, x \Leftarrow e$

Table 2. Simplification rules

In mixin sum (S), deferred components can be shared whereas for the other components disjoint union is performed (recall example 2 in Section 2). Note that, except for $\mathsf{DN}(\Delta_1) \cap \mathsf{DN}(\Delta_2)$, all other conditions can be satisfied by an appropriate α-conversion. The last condition avoids capture of free variables.
In (F), like in example 3, the deferred component X can be frozen only if X is also defined; then, the deferred component $x{:}\,X$ is deleted and the local component $x \lhd e$ is inserted, which means either $x \Leftarrow e$ if X is defined by $X \Leftarrow e$, or $x = e$ if X is defined by $X = e$. Furthermore $X \lhd e$ is transformed into $X = x$ since if X is computational, then e must be evaluated only once[3].
In (D), the defined component is simply removed, as in example 4.
Rule (A) expresses doall in terms of mdo: first, all computational components are evaluated according to the order given in the mixin (recall example 5), then a record value is returned containing both the non computational (o_1) and the computational defined components (o_2) of the mixin; substitution of the non computational local components (ρ) is needed in order to avoid variables to escape from their scope (the let construct is used because local variables can be mutually recursive). Finally, note that each computational defined component $X \Leftarrow e$ is transformed into $X = x_X$, with x_X freshly chosen variable, because e must be evaluated only once.
Simplification enjoys the Church Rosser and Subject Reduction properties.

Proposition 1 (CR for $\longrightarrow$). *The relation $\longrightarrow$ is confluent.*

[3] For simplicity, this transformation is always applied, even though is really needed only when X is computational.

Proof. The simplification rules are left-linear and non-overlapping. □

Proposition 2 (SR for $\longrightarrow$). *If $\Gamma \vdash_\Sigma e{:}\tau$ and $e \longrightarrow e'$, then $\Gamma \vdash_\Sigma e'{:}\tau$.*

Proof. By case analysis on the simplification rules. □

3.2 Computation

We now define *configurations* $Id \in \mathsf{Conf}$, that represent snapshots of the execution of a program, and the computation relation $\longmapsto$ (see Table 3), that describes how program execution evolves. Over these configurations we give an operational semantics that ensures the correct sequencing of computational effects, by adopting some well-established technique for specifying the operational semantics of programming languages (see [WF94]).

- Stores $\boxed{\mu \in \mathsf{S} \triangleq \mathsf{L} \overset{fin}{\rightarrow} \mathsf{E}}$ map locations to their content.
- Evaluation Contexts $\boxed{E \in \mathsf{EC}::= \Box \mid E[\mathsf{mdo}\,(x \Leftarrow \Box, \Theta; e)]}$ for terms of computational type.
- A configuration $\boxed{(\mu, e, E) \in \mathsf{Conf} \triangleq \mathsf{S} \times \mathsf{E} \times \mathsf{EC}}$ is a snapshot of the execution of a program: μ is the current store, e is the program fragment under consideration and E is the evaluation context for e.
- Bad terms b are terms that are stuck because they depend on a variable

$$\boxed{b \in \mathsf{BE}::= x \mid b.X \mid b + e \mid e + b \mid b!X \mid b \setminus X \mid \mathsf{doall}(b) \mid \mathsf{get}(b) \mid \mathsf{set}(b, e)}$$

- Computational Redexes r are terms that enable computation (with no need for simplification); when r is a bad term, we raise a run-time error.

$$\boxed{r \in \mathsf{R}::= \mathsf{mdo}\,(\Theta; e) \mid \mathsf{ret}(e) \mid \mathsf{new}(e) \mid \mathsf{get}(l) \mid \mathsf{set}(l, e) \mid b}$$

Definition 1. *The sets $\mathsf{CV}(E)$ and $\mathsf{FV}(E)$ of captured and free variables are*

- $\mathsf{CV}(\Box) \triangleq \mathsf{FV}(\Box) \triangleq \emptyset$
- $\mathsf{CV}(E[\mathsf{mdo}\,(x \Leftarrow \Box, \Theta; e)]) \triangleq \mathsf{CV}(E) \cup \{x\} \cup \mathsf{DV}(\Theta)$ *and*
  $\mathsf{FV}(E[\mathsf{mdo}\,(x \Leftarrow \Box, \Theta; e)]) \triangleq \mathsf{FV}(E) \cup (\mathsf{FV}(\Theta, e) \setminus \mathsf{CV}(E[\mathsf{mdo}\,(x \Leftarrow \Box, \Theta; e)]))$

Rules for monadic binding deserve some explanations. Rule (M.0) deals with the special case of empty binding; rule (M.1) starts the computation when the binding is not empty: the first expression of the binding is evaluated and renaming is needed in order to avoid clashes due to nested monadic bindings; rule (M.2) completes the computation of the binding variables: when the last variable has been computed, it can be substituted with its "value" (the let construct is used because of mutual recursion) in both the store and the body of mdo which now can be evaluated; finally, (M.3) is used for continuing the computation by considering the next binding variable and is similar to (M.2).
The confluent simplification relation $\longrightarrow$ on terms extends in the obvious way to a confluent relation (still denoted $\longrightarrow$) on stores, evaluation contexts, computational redexes and configurations.
A *complete program* corresponds to a closed term $e \in \mathsf{E}_0$ (with no occurrences of locations l), and its evaluation starts from the *initial configuration* $(\emptyset, e, \Box)$. The following properties ensure that only closed configurations are reachable (by $\longrightarrow$ and $\longmapsto$ steps) from the initial one.

Completion step

(done) $(\mu, \mathsf{ret}(e), \Box) \longmapsto \mathsf{done}$

Recursive monadic binding steps

(M.0) $(\mu, \mathsf{mdo}\,(\emptyset; e)\,, E) \longmapsto (\mu, e, E)$
(M.1) $(\mu, \mathsf{mdo}\,(x_1 \Leftarrow e_1, \Theta; e)\,, E) \longmapsto (\mu, e_1, E[\mathsf{mdo}\,(x_1 \Leftarrow \Box, \Theta; e)])$
with the variables in $\mathsf{DV}(x_1 \Leftarrow e_1, \Theta)$ renamed to avoid clashes with $\mathsf{CV}(E)$
(M.2) $(\mu, \mathsf{ret}(e_1), E[\mathsf{mdo}\,(x_1 \Leftarrow \Box; e)]) \longmapsto (\mu\{\rho\}, e\{\rho\}, E)$ where $\rho \stackrel{\Delta}{=} \{x_1 \colon \mathsf{let}(x_1 \colon e_1; x_1)\}$
(M.3) $(\mu, \mathsf{ret}(e_1), E[\mathsf{mdo}\,(x_1 \Leftarrow \Box, x_2 \Leftarrow e_2, \Theta; e)]) \longmapsto$
$(\mu\{\rho\}, e_2\{\rho\}, E[\mathsf{mdo}\,(x_2 \Leftarrow \Box, \Theta; e)\{\rho\}])$ where $\rho \stackrel{\Delta}{=} \{x_1 \colon \mathsf{let}(x_1 \colon e_1; x_1)\}$

Imperative steps

(I.1) $(\mu, \mathsf{new}(e), E) \longmapsto (\mu\{l \colon e\}, \mathsf{ret}(l), E)$ where $l \notin \mathsf{dom}(\mu)$
(I.2) $(\mu, \mathsf{get}(l), E) \longmapsto (\mu, \mathsf{ret}(e), E)$ provided $e = \mu(l)$
(I.3) $(\mu, \mathsf{set}(l, e), E) \longmapsto (\mu\{l \colon e\}, \mathsf{ret}(l), E)$ provided $l \in \mathsf{dom}(\mu)$

Error step caused by a bad term

(err) $(\mu, b, E) \longmapsto \mathsf{err}$

Table 3. Computation Relation

Lemma 1.

1. *If* $(\mu, e, E) \longrightarrow (\mu', e', E')$, *then* $\mathsf{dom}(\mu) = \mathsf{dom}(\mu')$, $\mathsf{CV}(E) = \mathsf{CV}(E')$, $\mathsf{FV}(\mu') \subseteq \mathsf{FV}(\mu)$, $\mathsf{FV}(e') \subseteq \mathsf{FV}(e)$ *and* $\mathsf{FV}(E') \subseteq \mathsf{FV}(E)$.
2. *If* $(\mu, e, E) \longmapsto (\mu', e', E')$ *and* $\mathsf{FV}(e, \mu) \subseteq \mathsf{CV}(E)$ *and* $\mathsf{FV}(E) = \emptyset$, *then* $\mathsf{FV}(e', \mu') \subseteq \mathsf{CV}(E')$, $\mathsf{FV}(E') = \emptyset$ *and* $\mathsf{dom}(\mu) \subseteq \mathsf{dom}(\mu')$.

Bad terms and computational redexes are closed w.r.t. simplification.

Lemma 2. *If* $b \longrightarrow e$, *then* $e \in \mathsf{BE}$. *If* $r \longrightarrow e$, *then* $e \in \mathsf{R}$.

When the program fragment under consideration is a computational redex, it is irrelevant whether simplification is done before or after a step of computation.

Theorem 1 (Bisimulation). *If* $(\mu_1, r_1, E_1) \stackrel{*}{\longrightarrow} (\mu_2, r_2, E_2)$, *then*

1. $(\mu_1, r_1, E_1) \longmapsto Id_1$ *implies* $\exists Id_2$ *s.t.* $(\mu_2, r_2, E_2) \longmapsto Id_2$ *and* $Id_1 \stackrel{*}{\longrightarrow} Id_2$
2. $(\mu_2, r_2, E_2) \longmapsto Id_2$ *implies* $\exists Id_1$ *s.t.* $(\mu_1, r_1, E_1) \longmapsto Id_1$ *and* $Id_1 \stackrel{*}{\longrightarrow} Id_2$

where Id_1 *and* Id_2 *range over* $\mathsf{Conf} \cup \{\mathsf{done}, \mathsf{err}\}$.

Proof. See [MF03]. □

3.3 Type safety

We go through the proof of type safety for CMS_{do}. The result is standard, but we make some adjustments to the Subject Reduction and Progress properties for $\Longrightarrow \stackrel{\Delta}{=} \longrightarrow \cup \longmapsto$, in order to stress the different role of simplification $\longrightarrow$ and computation $\longmapsto$. First of all, we define well-formedness for evaluation contexts $\Gamma, \Box \colon M\tau \vdash_\Sigma E \colon M\tau'$ (in Table 4) and configurations $\Gamma \vdash_\Sigma (\mu, e, E)$.

$$(\Box)\ \frac{}{\emptyset, \Box\colon M\tau \vdash_\Sigma \Box\colon M\tau}$$

$$(\text{mdo})\ \frac{\begin{array}{l}\{\Gamma, x_1\colon \tau_1, \Gamma_\Theta \vdash_\Sigma e'\colon M\tau' \mid (x' \Leftarrow e') \in \Theta \wedge \tau' = \Gamma_\Theta(x')\} \\ \Gamma, x_1\colon \tau_1, \Gamma_\Theta \vdash_\Sigma e\colon M\tau_2 \\ \Gamma, \Box\colon M\tau_2 \vdash_\Sigma E\colon M\tau\end{array}}{\Gamma, x_1\colon \tau_1, \Gamma_\Theta, \Box\colon M\tau_1 \vdash_\Sigma E[\mathsf{mdo}\,(x_1 \Leftarrow \Box, \Theta; e)]\colon M\tau}\ \mathsf{dom}(\Gamma_\Theta) = \mathsf{DV}(\Theta)$$

Table 4. Well-formed evaluation contexts

Definition 2 (Well-formed configurations). $\Gamma \vdash_\Sigma (\mu, e, E) \overset{\Delta}{\Longleftrightarrow}$

- $\mathsf{dom}(\Sigma) = \mathsf{dom}(\mu)$ *and* $\mathsf{dom}(\Gamma) = \mathsf{CV}(E)$*;*
- $\mu(l) = e_l$ *and* $\Sigma(l) = \tau_l$ *imply* $\Gamma \vdash_\Sigma e_l\colon \tau_l$*;*
- *exists* τ *such that* $\Gamma \vdash_\Sigma e\colon M\tau$ *derivable;*
- *exists* τ' *such that* $\Gamma, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ *derivable (see Table 4).*

The formation rules of Table 4 for deriving $\Gamma, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ ensure that

- Γ assigns a type to all captured variables of E, indeed $\mathsf{dom}(\Gamma) = \mathsf{CV}(E)$;
- E has no free variables and cannot capture a variable x twice.

Proposition 3 (SR).

1. *If* $\Gamma \vdash_\Sigma (\mu, e, E)$ *and* $(\mu, e, E) \longrightarrow (\mu', e', E')$*, then* $\Gamma \vdash_\Sigma (\mu', e', E')$*.*
2. *If* $\Gamma \vdash_\Sigma (\mu, e, E)$ *and* $(\mu, e, E) \longmapsto (\mu', e', E')$*, then there exist* $\Sigma' \supseteq \Sigma$ *and* Γ' *compatible with* Γ *such that* $\Gamma' \vdash_{\Sigma'} (\mu', e', E')$*.*

Theorem 2 (Progress). *If* $\Gamma \vdash_\Sigma (\mu, e, E)$*, then one of the following cases holds*

1. $e \in \mathsf{R}$ *and* $(\mu, e, E) \longmapsto$ *, or*
2. $e \notin \mathsf{R}$ *and* $e \longrightarrow$

Proof. See the long version of this paper available on the web. □

4 Related Work

The notion of mixin module was firstly introduced in Bracha's PhD thesis [Bra92] as a generalization of the notion of mixin class (see for instance [BC90]). The semantics of the mixin language in [Bra92] is based on the early work on denotational semantics of inheritance [Coo89,Red88] and is defined by a translation into an untyped λ-calculus equipped with a fixpoint operator and a rather rich set of record operators. Furthermore, imperative features are only marginally considered by implicitly using the technique developed in [Hen93] for extending the semantics of inheritance given in [Coo89,Red88] to object-oriented languages with state.
After this pioneer work, some proposals for extending existing languages with a system of mixin modules were considered: [DS96] and [FF98a,FF98b] go in this direction; however, imperative features are not considered and recursion problems are solved by separating initialization from component definition.
The first calculi based on the notion of mixin modules appeared in [AZ99,AZ02] and then in [WV00,MT00], but all of them are defined in a purely functional setting. More recently, [HL02] has considered a *CMS*-like calculus, called CMS_v, with a refined type system in order to avoid

bad recursion in a call-by-value setting. A separate compilation schema for CMS_v has been also investigated by means of a translation down to a call-by-value λ-calculus λ_B extended with a non-standard `let rec` construct, inspired by the calculus defined in [Bou02].
Like CMS_{do}, both λ_B and the calculus of Boudol serve as semantic basis for programming languages supporting mixins and introduce non-standard constructs for recursion which can produce terms having an undefined semantics. However, λ_B does not have imperative features, whereas the calculus in [Bou02] does not allow recursion in the presence of side-effects. For instance, in CMS_{do} the term $\mathsf{mdo}\,(x \Leftarrow \mathsf{new}(x); \mathsf{ret}(x))$ has a well-defined semantics, whereas the corresponding translated term $\mathsf{let\ rec}\ x = \mathsf{ref}\ x\ \mathsf{in}\ x$ in Boudol's calculus is not well-typed; indeed, the evaluation of this term gets stuck. Another advantage of our approach is that the separation of concerns made possible by the monadic metalanguage allows us to retain the equational reasoning of *CMS*.
On the other hand, the more refined type systems adopted in [HL02,Bou02] are able to statically detect all bad recursive declarations.
As already mentioned, the definition of the mdo construct in CMS_{do} is inspired by the work on the semantics of recursive monadic bindings in Haskell [EL00,ELM01,ELM02,EL02]. Our semantics is partly related to that in [ELM01], however the notion of heap in our calculus has been made implicit (thanks to the `let rec` construct), since we are interested in a more abstract approach; and furthermore, the recursive `do` in [EL02] does not perform an incremental binding as happens in our semantics, but rather all values are bound to the corresponding variables only after all computations in the recursive monadic binding have been evaluated.

5 Conclusion and Future Work

We have defined CMS_{do}, a monadic mixin calculus in which mixin modules can contain components of three kinds: defined (bound to an expression), deferred (declared but not yet defined) and computational (bound to a computation which must be performed before actually using the module for component selection). Mixin modules can be combined by the sum, freeze and restrict operators of *CMS*; moreover, a `doall` operator triggers all the computations in a mixin module.
We have provided a simple type system for the language, a simplification relation defined by local rewrite rules with no side-effects (satisfying the CR and SR properties), and a computation relation which models global evaluation able to modify the store (satisfying the SR property). Moreover, we have stated a bisimulation result (simplification does not affect computation steps) and the progress property for the combined relation; however, errors due to bad recursive declarations are only dynamically detected, since here we have preferred to keep a simple type system.
We envisage at least two possibilities which deserve investigation in the direction of defining more refined type systems. First, the dynamic errors due to bad recursive declarations mentioned above could be detected by introducing a type system similar to that in [HL02,Bou02] keeping explicit trace of *dependencies* between the evaluation of two computational components. On a different side, a type system distinguishing between modules possibly containing some computational components (or variables) and those with no computational components (and variables), would allow selection on *CMS* mixins, so that *CMS* could be more directly embedded into CMS_{do}.
For what concerns applications, CMS_{do} can be considered a powerful kernel calculus allowing to express, on one side, a variety of different operators for combination of software modules (including linking, parameterized modules as ML functors, overriding in the sense of object-oriented languages, see [AZ02] for details), on the other side different choices in the evaluation of computations. In particular, we mention at least two relevant scenarios of application: the modeling of object-oriented features, including the difference between computations which must be performed before instantiating a class, as field initializers, and computations which are evaluated each time

they are selected, as methods; and the possibility of expressing different policies for dynamic linking and verification.

References

[AZ99] D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Principles and Practice of Declarative Programming, 1999*, number 1702 in Lecture Notes in Computer Science, pages 62–79. Springer Verlag, 1999.

[AZ02] D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, March 2002.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, October 1990.

[Bou02] G. Boudol. The recursive record semantics of objects revisited. To appear in *Journal of Functional Programming*, 2002.

[Bra92] G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.

[Coo89] W.R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Dept. of Computer Science, Brown University, 1989.

[DS96] D. Duggan and C. Sourelis. Mixin modules. In *Intl. Conf. on Functional Programming*, Philadelphia, May 1996. ACM Press.

[EL00] L. Erkök and J. Launchbury. Recursive monadic bindings. In *Intl. Conf. on Functional Programming 2000*, pages 174–185, 2000.

[EL02] L. Erkök and J. Launchbury. A recursive do for Haskell. In *Haskell Workshop'02*, pages 29–37, 2002.

[ELM01] L. Erkök, J. Launchbury, and A. Moran. Semantics of $fixIO$. In *FICS'01*, 2001.

[ELM02] L. Erkök, J. Launchbury, and A. Moran. Semantics of value recursion for monadic input/output. *Journal of Theoretical Informatics and Applications*, 36(2):155–180, 2002.

[FF98a] R.B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Intl. Conf. on Functional Programming 1998*, September 1998.

[FF98b] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *PLDI'98 - ACM Conf. on Programming Language Design and Implementation*, pages 236–248, 1998.

[Hen93] A. V. Hense. Denotational semantics of an object-oriented programming language with explicit wrappers. *Formal Aspects of Computing*, 5(3):181–207, 1993.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, 1994.

[HL02] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, editor, *ESOP 2002 - Programming Languages and Systems*, number 2305 in Lecture Notes in Computer Science, pages 6–20. Springer Verlag, 2002.

[Ler94] X. Leroy. Manifest types, modules and separate compilation. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[MF03] E. Moggi and S. Fagorzi. A Monadic Multi-stage Metalanguage. In A.D. Gordon, editor, *Foundations of Software Science and Computational Structures - FOSSACS 2003*, volume 2620 of *LNCS*, pages 358–374. Springer Verlag, 2003.

[MT00] E. Machkasova and F.A. Turbak. A calculus for link-time compilation. In G. Smolka, editor, *ESOP 2000 - Programming Languages and Systems*, number 1782 in Lecture Notes in Computer Science, pages 260–274, Berlin, 2000. Springer Verlag.

[Red88] U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 289–297, 1988.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WV00] J.B. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In G. Smolka, editor, *ESOP 2000 - Programming Languages and Systems*, number 1782 in Lecture Notes in Computer Science, pages 412–428, Berlin, 2000. Springer Verlag.