A CALCULUS WITH LAZY MODULE OPERATORS

Davide Ancona, Sonia Fagorzi and Elena Zucca
DISI - Università di Genova
Via Dodecaneso, 35, 16146 Genova (Italy)[*]
{davide,fagorzi,zucca}@disi.unige.it

Abstract Modern programming environments such as those of Java and C# support dynamic loading of software fragments. More in general, we can expect that in the future systems will support more and more forms of interleaving of *reconfiguration* steps and standard *execution* steps, where the software fragments composing a program are dynamically changed and/or combined on demand and in different ways. However, existing kernel calculi providing formal foundations for module systems are based on a *static* view of module manipulation, in the sense that open code fragments can be flexibly combined together, but all module operators must be performed once for all *before* starting execution of a program, that is, evaluation of a module component.

The definition of clean and powerful module calculi supporting *lazy* module operators, that is, operators which can be performed *after* the selection of some module component, is still an open problem. Here, we provide an example in this direction (the first at our knowledge), defining CMS^{ℓ}, an extension of the Calculus of Module Systems [5] where module operators can be performed at execution time and, in particular, are executed on demand, that is, only when needed by the executing program. In other words, execution steps, if possible, take the precedence over reconfiguration steps. The type system of the calculus, which is proved to be sound, relies on a dependency analysis which ensures that execution will never try to access module components which cannot become available by performing reconfiguration steps.

Keywords: module calculi, dynamic linking

1 Introduction

In the last years considerable effort has been invested in studying theoretical foundations and designing advanced forms of module systems [5, 15, 13, 12, 2], inspired by the unifying principle of two separate linguistic levels, a *module language* providing operators for combining software components, with their own typing rules, constructed on top of a *core language* for defining module components. In particular, module calculi such as *CMS* (Calculus of Module Systems) [5] provide a simple and powerful model allowing to express a large variety of existing mechanisms for com-

*Partially supported by Dynamic Assembly, Reconfiguration and Type-checking - EC project IST-2001-33477, and APPSEM II - Thematic network IST-2001-38957.

bining software components, hence can be used as a paradigmatic calculus for modular languages, in the same spirit the lambda calculus is used for functional programming. Indeed, modules in *CMS* are constructed from *basic modules* (of the form $[\iota;\ o;\ \rho]$ where ι, o and ρ model input, output and local components, respectively) by only three primitive operators: *sum* (merging two modules), *link* (called *freeze* in the original formulation in [5], binding an input to an output component) and *reduct* (independently renaming input and output components). As shown in previous papers [5, 4], these operators allow to express, e.g., parameterized modules similar to ML functors, extension with overriding as in object-oriented programming, and mixin modules. However, *CMS* (as other module calculi as well) is based on a *static* view of module manipulation, in the sense that open code fragments can be flexibly combined together, but before starting program execution we must eventually obtain a fully reduced, closed module.
This is formally reflected in *CMS* by the fact that *selection,* denoted *M.X,* where *M* is a module expression and *X* is the name of a module component, can only be performed when *M* has form $[\,;\ o;\ \rho]$, that is, is a basic module (no module operators remain) and, moreover, has no input components. In other words, before actually *using* a module, all *configuration* steps (that is, those concerning assembly and manipulation of code fragments) must have been performed, hence in particular all the component names must have been resolved (that is, no dependency on other fragments is allowed),
However, in widely-used programming environments, such as those of Java and C#, single code fragments are dynamically linked to an already executing program. More generally, we can expect that in the future systems will support more and more forms of interleaving of *reconfiguration* steps and standard *execution* steps, where the software fragments composing a program are dynamically changed and/or combined on demand and in different ways. To our knowledge, only a little amount of literature exists on this subject, mainly concerned with the modeling of concrete mechanisms in existing programming environments (see, e.g., the large amount of work of Drossopoulou and others on phases of dynamic linking and verification in Java-like languages [9, 10]).
In particular, what is still missing is the definition of clean and powerful module calculi supporting *lazy* module operators, that is, operators which can be performed *after* the evaluation of some module component has started, hence providing formal foundations for systems where reconfiguration and standard execution steps are interleaved (as *CMS* or other module calculi do for static module manipulation).
Here, we provide a proposal in this direction (the first to our knowledge), defining CMS^{ℓ}, an extension of *CMS* where module operators can be performed at execution time and, in particular, are executed *on demand,* that is, only when needed by the executing program. In other words, execution steps, if possible, take the precedence over reconfiguration steps.
The type system of the calculus, which is proved to be sound, relies on a dependency analysis which ensures that execution never tries to access module components which cannot become available by performing reconfiguration steps.
The rest of the paper is organized as follows. In Sect.2 we briefly revise the original *CMS* and then informally introduce CMS^{ℓ} by some examples illustrating the new possibilities offered in this calculus. In Sect.3 we give the syntax and the reduction rules, in Sect.4 the type system and in Sect.5 the results (confluence and soundness). Finally, in the Conclusion we summarize the contribution of the paper, compare the

approach here with our previous work on dynamic linking [2, 11] and describe further work.

2 An informal introduction

In this section we briefly introduce *CMS* and then illustrate the new possibilities offered by CMS^{ℓ} by some examples, written by using some syntactic sugar.
A *CMS basic module* consists of *output* and *local* components, bound to an expression, and *input* components, declared but not yet defined. For instance,

```
module M1 is
 import X as x, export Y = e1[x,y], local  y = e2[x,y]
end M1
```

is a basic module with one input, one output and one local component, where `e1[x,y]` and `e2[x,y]` denote two arbitrary expressions possibly containing `x` and `y` as free variables. Note that input components are associated with both a name (as `X`) and a variable (as `x`); component names are used for accessing input and output components from the outside, while variables are used for accessing input and local components from inside the module. Local components are not visible from outside and can be mutually recursive.
Two modules can be combined by the *sum* operation, which performs the union of the input components (in the sense that components with the same name are shared), and the disjoint union of the output and local components. However, while the sets of output names must be disjoint, the disjoint union of local components can always be performed (using **α-renaming** of local variables when needed).
For instance, below module `M3` is defined as the sum of `M1` above and another basic module `M2`.

```
module M2 is
  import Y as y,  export X = e3[x,y],  local  x = e4[x,y]
end M2
```

Module `M3 = M1 + M2` simplifies to

```
module import X  as x, Y as y',
       export Y  = e1[x,y], X  = e3[x',y'],
       local  y = e2[x,y], x' = e4[x',y']
end
```

Note that the sum operation supports cross-module recursion: in module `M3`, the definition of `X` is needed by `M1` and is provided by `M2`, whereas the definition of `Y` is needed by `M2` and is provided by `M1`. However, in the sum above there is no connection yet between input and output names; this can be accomplished by means of the link operator described below.
The *link* operation connects input and output components having the same name inside a module, so that an input component becomes local. For instance, in

```
link X in
 (module import X  as x, export X  = e[x, ...], local  ... end)
```

which simplifies to

```
module export X = e[x, ...], local  ..., x = e[x, ...] end
```

the input name X has been effectively bound to the corresponding output component. The *reduct* operator performs a renaming of component names where input and output names are renamed independently. The input renaming is a mapping whose domain and codomain are old input names and new input names, respectively , whereas the output renaming is a mapping whose domain and codomain are new output names and old output names, respectively. For instance,

```
rename input X by X, X2 by X,  _ by X', output Y1 by Y, Y2 by Y
in module
  import X1 as x1, X2 as x2,
  export Y = e1[x1,x2,x], Y' = e2[x1,x2,x],
  local x = e[x1,x2,x]
end
```

simplifies to

```
module
  import X as x1,  X as x2,  X' as x',
  export Y1 = e1[x1,x2,x], Y2 = e1[x1,x2,x],
  local x = e[x1,x2,x]
end
```

Note that the two renamings can be non-injective and non-surjective. A non-injective input renaming allows to merge two input names (in the example X1 and X2 in X), whereas a non-surjective is used for adding dummy input names (X' in the example). A non-injective output renaming allows duplication of definitions (in the example the definition of Y is used as definition of both Y1 and Y2), whereas a non-surjective one is used for deleting output components (Y' in the example).
In the following examples M\Y denotes the application to the module M of a reduct operator s.t. the input renaming is the identity and the output renaming is the embedding of all output names of M except Y in themselves. In other words, M\Y denotes the module where the Y component has been deleted.
Output components can be accessed from the outside by means of the selection operator. In CMS^ℓ, selection is much more general than in *CMS,* where it can be performed only on basic modules with no input components.
Consider, for instance, the following configuration:

```
C = (module import X as x, export Y = e[x], local  ... end).Y
```

This configuration is well-formed in CMS^ℓ if the defining expression e of Y does not use the variable x, which is bound to an input component. Moreover, even in the case e uses the variable x, we can obtain a well-formed CMS^ℓ configuration by inserting *C* in a context where the input component X can become available, as shown below:

```
link X in (C  + module import ... export X = ... local ...)
```

The following examples illustrate how CMS^ℓ allows dynamic reconfiguration of systems. First we show how CMS^ℓ lazy sum and link operators allow to model loading of software on demand. Consider a situation where there is a program Prg to be executed, possibly requiring other software fragments located in different sites, e.g., on the web. In *CMS,* this can be modeled by the following module expression, where each basic module in the sum expression intuitively corresponds to software from a different site.

```
(link X, Y in (
  module import X as x, Y as y, export Prg = e,  local  ... end +
  module import Y as y, export X = ..., local  ...  end  +
  module import X as x,  export Y = ..., local  ... end)
).Prg
```

In *CMS*, in order to select the `Prg` output component, the module expression must be first of all reduced to a basic module, regardless of the nature of the defining expression of `Prg`. Hence the program can be executed only after loading and combining software from all sites, thus requiring a not negligible amount of time. In CMS^{ℓ} the situation described above could be modeled instead by the term:

```
link X, Y in (
(module import X as x, Y as y, export Prg = e, local  ... end).Prg) +
 module import Y as y, export X = ...,  local  ... end  +
 module import X as y, export Y = ..., local  ... end)
```

In this case, execution of the program can start immediately, and the sum and link will be performed only if and when the evaluation of the expression `e` will need `x` and `y`. The following example shows how CMS^{ℓ} lazy reduct operator allows to express reference to different versions of the same software fragment. Consider a situation where two versions of a component `Y` are available.

```
link Y in (
 (module import Y as x, export Y = e,  local  ... end).Y  \ Y +
 module  import ..., export Y = e', ...,  local  ... end)
```

The old definition `e` of `Y` is initially selected and its evaluation starts. However, if during evaluation of `e` the variable `x` is needed, then reconfiguration steps are performed and the new definition `e'` of `Y` is used. However, note that only a limited form of dynamic reconfiguration is allowed, since all reconfiguration steps are planned statically: the fact that they will be actually performed depends on the program execution (thus allowing in particular to use different versions of a component at different stages, as shown above), but it is not possible to perform *different* reconfiguration steps depending on the execution. See the Conclusion for more comments on this point.

3 Syntax and Semantics

Notations We denote by $A \xrightarrow{fin} B$ the set of the partial functions f from A to B with finite domain, written $\mathsf{dom}(f)$; the image of f is written $\mathsf{img}(f)$. We denote by f, g the union of two partial functions with disjoint domain, whereas we use the notation $f \cup g$ for the union of two compatible partial functions, that is, s.t. $f(x) = g(x)$ for all $x \in \mathsf{dom}(f) \cap \mathsf{dom}(g)$. Finally, $\circ$ denotes composition of partial functions.

The syntax of the calculus is given in Fig. 1. We assume an infinite set Name of *names* X, an infinite set Var of *variables* x, and a set Exp of (core) expressions (the expressions of the underlying language used for defining module components). Indeed, as *CMS*, CMS^{ℓ} is a parametric and stratified calculus, which can be instantiated over different core calculi satisfying some (standard) assumptions specified in the sequel. In CMS^{ℓ}, however, differently from *CMS*, module components cannot be modules. Intuitively, names are used to refer to a module from the outside (hence they are used in reconfiguration steps), while variables are used to refer to a (basic) module from a program executing in the context of the components offered by this module.

$C \in \mathsf{Conf}$	$::=$			**configurations**
		$< [\iota; o; \rho]\,,\, e >$	$(\mathsf{dom}(\iota) \cap \mathsf{dom}(\rho) = \emptyset)$	executable basic conf.
	$\mid$	$C + M$		sum
	$\mid$	$\mathsf{link}_\sigma C$		link
	$\mid$	$_{\sigma^\iota}\lvert C \rvert_{\sigma^o}$		reduct
	$\mid$	$M.X$		selection
	$\mid$	M		non-executable conf.
$M \in \mathsf{MExp}$	$::=$			
	$\mid$	$[\iota; o; \rho]$	$(\mathsf{dom}(\iota) \cap \mathsf{dom}(\rho) = \emptyset)$	non-executable basic conf.
	$\mid$	$M + M$		sum
	$\mid$	$\mathsf{link}_\sigma M$		link
	$\mid$	$_{\sigma^\iota}\lvert M \rvert_{\sigma^o}$		reduct
$e \in \mathsf{Exp}$	$::=$	$x \mid \ldots$		**(core) expressions**
$\tau \in \mathsf{Type}$	$::=$	$\ldots$		**(core) types**

$\iota : \mathsf{Var} \xrightarrow{fin} \mathsf{Name}$	**input assignments**
$o : \mathsf{Name} \xrightarrow{fin} \mathsf{Exp}$	**output assignments**
$\rho : \mathsf{Var} \xrightarrow{fin} \mathsf{Exp}$	**local assignments**
$\sigma : \mathsf{Name} \xrightarrow{fin} \mathsf{Name}$	renamings

Figure 1. Syntax

This distinction between names and variables is standard in module calculi and, besides the methodological motivation explained above, has technical motivations as well, such as allowing α-conversion for variables while preserving external interfaces (see, e.g., [5] for an extended discussion of this point).
Terms of the calculus are called *configurations.* Configurations can be either *non-executable* configurations (module expressions) M, or *executable* configurations, which are constructed from *executable basic configurations* by the three primitive module operators *sum, link* and *reduct.* Moreover, a configuration can be obtained by selecting a component from a module expression.
An executable basic configuration is a pair $< [\iota; o; \rho]\,,\, e >$, consisting of a basic module and a core expression. Basic modules are as in *CMS* and consist of three components. The ι component is a mapping from variables to names and represents the *input* interface of the module; the o component is a mapping from names into expressions and represents the *output* interface of the module; the ρ component is a mapping from variables into expressions and represents the local (that is, already linked) components. Variables in the domain of ι and ρ are called the *deferred* and the *local* variables of the basic module, respectively.
Basic (both executable and non-executable) configurations are well-formed only if the sets of deferred and local variables are disjoint.
We will explain module operators in more detail when introducing reduction rules.
Expressions of the core language are not specified; we only assume that they contain variables. For the examples in the sequel we assume that core expressions contain integer constants and the usual operations on integers.
In Fig.2 and Fig.3 we give the reduction rules of the calculus. For convenience, we first give the reduction rules for non-executable configurations (module expressions) and then those for executable configurations. By definition, the one step reduction relation $\longrightarrow$ is the relation over well-formed terms inductively defined by the rules. For this

$$(M\text{-sum})\ \frac{}{M_1 + M_2 \longrightarrow [\iota_1, \iota_2;\ o_1, o_2;\ \rho_1, \rho_2]}\ \begin{array}{l} M_i \equiv [\iota_i;\ o_i;\ \rho_i]\,, i \in \{1,2\} \\ \mathsf{BV}\,(M_1) \cap \mathsf{FV}(M_2) = \emptyset \\ \mathsf{BV}\,(M_2) \cap \mathsf{FV}(M_1) = \emptyset \end{array}$$

$$(M\text{-link})\ \frac{}{\mathsf{link}_\sigma[\iota_1, \iota_2;\ o;\ \rho] \longrightarrow [\iota_2;\ o;\ \rho, o \circ \sigma \circ \iota_1]}\ \mathsf{img}(\iota_2) \cap \mathsf{dom}(\sigma) = \emptyset$$

$$(M\text{-reduct})\ \frac{}{{}_{\sigma^\iota}|[\iota;\ o;\ \rho]|_{\sigma^o} \longrightarrow [\sigma^\iota \circ \iota;\ o \circ \sigma^o;\ \rho]}$$

Figure 2. **Reduction rules for non-executable configurations (module expressions)**

reason, we have omitted all side conditions ensuring well-formedness of terms, since those are satisfied by definition.

Reduction rules for sum, link and reduct on non-executable configurations are exactly those for module expressions in *CMS*. We give here a brief description, referring to [5] for more detailed comments.

Sum The sum operation simply has the effect of gluing together two modules. The two explicit side conditions avoid undesired captures of free variables; $\mathsf{BV}\,([\iota;\ o;\ \rho])$ denotes the binding variables of $[\iota;\ o;\ \rho]$, that is, $\mathsf{dom}(\iota) \cup \mathsf{dom}(\rho)$, whereas $\mathsf{FV}([\iota;\ o;\ \rho])$ denotes the free variables (the definition of $\mathsf{FV}(e)$ depends on the core calculus) of $[\iota;\ o;\ \rho]$, that is, $(\bigcup_{X \in \mathsf{dom}(o)} \mathsf{FV}(o(X)) \cup \bigcup_{x \in \mathsf{dom}(\rho)} \mathsf{FV}(\rho(x))) \setminus \mathsf{BV}\,([\iota;\ o;\ \rho])$. Since the reduction is defined only over well-formed terms, the deferred and local variables of one module must be disjoint from those of the other (implicit side condition). Both the explicit and implicit side conditions above can always be satisfied by an appropriate α-conversion. For the same reason of well-formedness, the output names of the two modules must be disjoint (implicit side condition due to the fact that o_1, o_2 must be well-defined[1]); however, in this case the reduction gets stuck since this conflict cannot be resolved by an α-conversion. The only way to solve this problem is to explicitly rename the output names in an appropriate way by means of the reduct operator (see below), thus changing the term.

Link The link operator is essential for binding input with output in order to accomplish inter-connection of modules. A renaming σ explicitly specifies how resolution has to be performed, associating output to input names; the domain of σ can be a proper subset of all input names of the module so that resolution is partial

The effect of applying the link operator is that all input names that are resolved, represented by the set $\mathsf{img}(\iota_1)$, disappear and all the deferred variables mapped into them, represented by the set $\mathsf{dom}(\iota_1)$, become local. These variables are associated with the definition of the output component to which their associated (by ι_1) names are bound by σ, that is, $o(\sigma(\iota_1(x)))$, for all $x \in \mathsf{dom}(\iota_1)$). The composition is well-defined if the following implicit side conditions (needed for composition of mappings to be well-defined) are verified: $\mathsf{img}(\iota_1) \subseteq \mathsf{dom}(\sigma)$ and $\mathsf{img}(\sigma) \subseteq \mathsf{dom}(o)$. Note that this implies that only variables for which actually *exist* a corresponding definition become

[1]Note that, since ι goes "backwards", that is, from variables into names, the fact that ι_1, ι_2 must be well-formed does not prevent to share input names, but only to share deferred variables, what can be avoided by α-conversion.

Evaluation contexts

$$\begin{array}{lcl}
\mathcal{C}[\,] \in \mathsf{ECCtx} & ::= & \square \mid \mathcal{M}[\ ].X \mid \mathcal{C}[\,] + M \mid C + \mathcal{M}[\ ] \mid \mathsf{link}_\sigma \mathcal{C}[\,] \mid {}_{\sigma^\iota}|\mathcal{C}[\,]|_{\sigma^o} \\
\mathcal{M}[\ ] \in \mathsf{NCCtx} & ::= & \square \mid \mathcal{M}[\ ] + M \mid M + \mathcal{M}[\ ] \mid \mathsf{link}_\sigma \mathcal{M}[\ ] \mid {}_{\sigma^\iota}|\mathcal{M}[\ ]|_{\sigma^o} \\
\mathcal{E}[\,] \in \mathsf{ECtx} & ::= & \square \mid \ldots
\end{array}$$

$$(\mathcal{C}[\,]\text{-ctx})\ \frac{C \longrightarrow C'}{\mathcal{C}[C] \longrightarrow \mathcal{C}[C']} \qquad (\mathcal{M}[\ ]\text{-ctx})\ \frac{M \longrightarrow M'}{\mathcal{M}[M] \longrightarrow \mathcal{M}[M']}$$

$$(\text{core})\ \frac{e \underset{e}{\longrightarrow} e'}{<[\iota;\, o;\, \rho]\,,\, e> \ \longrightarrow\ <[\iota;\, o;\, \rho]\,,\, e'>}$$

$$(\text{var})\ \frac{}{<[\iota;\, o;\, \rho]\,,\, \mathcal{E}[x]> \ \longrightarrow\ <[\iota;\, o;\, \rho]\,,\, \mathcal{E}[\rho(x)]>}\ \begin{array}{l} x \in \mathsf{dom}(\rho) \\ \mathcal{E}[x] \underset{e}{\not\longrightarrow} \end{array}$$

$$(C\text{-link})\ \frac{\mathsf{link}_\sigma[\iota;\, o;\, \rho] \longrightarrow [\iota';\, o';\, \rho']}{\mathsf{link}_\sigma <[\iota;\, o;\, \rho]\,,\, \mathcal{E}[x]> \ \longrightarrow\ <[\iota';\, o';\, \rho']\,,\, \mathcal{E}[x]>}\ \begin{array}{l} x \in \mathsf{dom}(\iota) \\ \mathcal{E}[x] \underset{e}{\not\longrightarrow} \end{array}$$

$$(C\text{-sum})\ \frac{[\iota_1;\, o_1;\, \rho_1] + [\iota_2;\, o_2;\, \rho_2] \longrightarrow [\iota;\, o;\, \rho]}{<[\iota_1;\, o_1;\, \rho_1]\,,\, \mathcal{E}[x]> + [\iota_2;\, o_2;\, \rho_2] \ \longrightarrow\ <[\iota;\, o;\, \rho]\,,\, \mathcal{E}[x]>}\ \begin{array}{l} x \in \mathsf{dom}(\iota) \\ \mathcal{E}[x] \underset{e}{\not\longrightarrow} \end{array}$$

$$(C\text{-reduct})\ \frac{{}_{\sigma^\iota}|[\iota;\, o;\, \rho]|_{\sigma^o} \longrightarrow [\iota';\, o';\, \rho']}{{}_{\sigma^\iota}|<[\iota;\, o;\, \rho]\,,\, \mathcal{E}[x]>|_{\sigma^o} \ \longrightarrow\ <[\iota';\, o';\, \rho']\,,\, \mathcal{E}[x]>}\ \begin{array}{l} x \in \mathsf{dom}(\iota) \\ \mathcal{E}[x] \underset{e}{\not\longrightarrow} \end{array}$$

$$(\text{sel})\ \frac{}{[\iota;\, o;\, \rho].X \ \longrightarrow\ <[\iota;\, o;\, \rho]\,,\, o(X)>}\ X \in \mathsf{dom}(o)$$

Figure 3. Reduction rules for executable configurations

local, thus ensuring that we cannot create modules containing undefined (that is, neither local nor deferred) variables. The explicit side condition just ensures that $\mathsf{img}(\iota_1)$ actually contains *all* the input names that have to be resolved as specified by σ.

Reduct The reduct operator performs a renaming of component names and does not change the local assignment and the variables of a module; its effect is simply a composition of maps which can be correctly performed only if $\mathsf{img}(\iota) \subseteq \mathsf{dom}(\sigma^\iota)$ and $\mathsf{img}(\sigma^o) \subseteq \mathsf{dom}(o)$ (implicit side condition). Note that input and output names are renamed independently, and that the two renamings can be non-injective and non-surjective. A non-injective map σ^ι allows sharing of input names, whereas a non-surjective one is used for adding dummy (in the sense that no variable is associated with them) input names; a non-injective map σ^o allows duplication of definitions, whereas a non-surjective map is used for hiding output components.
We describe now reduction rules for executable configurations,
The first two rules are the usual contextual closures for executable and non-executable configurations, respectively.
Rule (core) models an execution step which is an evaluation step of the core expression in the basic executable configuration (we denote by $\underset{e}{\longrightarrow}$ the reduction relation of the core calculus).

Rule (var) models the situation where the evaluation of the core expression needs a variable which has a corresponding definition in the current basic module (that is, is local). In this case, the evaluation can proceed by simply replacing the variable by its defining expression. Here and in the following rules, the side condition $\mathcal{E}[x] \not\xrightarrow[e]{}$ expresses the fact that evaluation at the core level is stuck. Note that this ensures that there is no overlapping between (core) steps and other steps, but of course does not prevent non-determinism inherited from the core level. For instance, assuming that the core expression in a configuration is $x+y$, and both x, y are local variables, variable x will be first considered for application of another rule if $x+y$ can only be seen as $\mathcal{E}[x]$ by the core context formation rules, whereas either x or y will be non-deterministically consideredif $x+y$ can be seen as both $\mathcal{E}[x]$ and $\mathcal{E}'[y]$.
The following three rules express the fact that, whenever the evaluation of the core expression needs a variable which has no corresponding definition in the module (that is, is deferred), then a *reconfiguration* step happens: more precisely, the innermost enclosing module operator is applied.
As combined effect of the above rules, execution proceeds by standard execution steps ((core) and (var) rules) until a deferred variable is encountered; in this case, reconfiguration steps are performed (from the innermost to the outermost module operator) until the variable becomes local and rule (var) can be applied.

EXAMPLE 1 *Let us write* $a_1 : b_1, \ldots, a_n : b_n$ *for the partial function mapping* a_i *to* b_i *for all* $i \in 1..n$ *(where the* a_i *must be different).*
$C \triangleq \mathsf{link}_{X:Y}(< [x : X; ; y : 1], y + x > + [; Y : 2;])$
$\longrightarrow \mathsf{link}_{X:Y}(< [x : X; ; y : 1], 1 + x > + [; Y : 2;])$
$\longrightarrow \mathsf{link}_{X:Y} < [x : X; Y : 2; y : 1], 1 + x > \longrightarrow < [; Y : 2; y : 1, x : 2], 1 + x >$
$\longrightarrow < [; Y : 2; y : 1, x : 2], 1 + 2 > \longrightarrow < [; Y : 2; y : 1, x : 2], 3 > \triangleq C'$

Note that this precedence of standard execution over reconfiguration only applies to the module operators which contain the executable configuration, whereas the remaining module operators can be evaluated non deterministically at each time during execution. However, this non determinism does not affect confluence, as will be proved in Sect.5 (Prop.3).

EXAMPLE 2 *Set* $M = {}_{\emptyset}|\mathsf{link}_{Z:U}[z : Z; U : 3;]|_{W:U}$ *and C as in Example 1. We have* $M \longrightarrow {}_{\emptyset}|[; U : 3; z : 3]|_{W:U} \longrightarrow [; W : 3; z : 3] \triangleq M'$ *and* $C{+}M \overset{*}{\longrightarrow} C'+M'$ *where all the reduction steps (included those in Example 1) can be arbitrarily interleaved.*

4 Type system

The type system of the calculus is given in Fig.4 and Fig.5.
The typing judgment for module expressions has form $\vdash_M M : [\pi^\iota; \pi^o; \mathcal{D}]$, meaning that M is a well-formed module expression of type $[\pi^\iota; \pi^o; \mathcal{D}]$. Types for module expressions are triples $[\pi^\iota; \pi^o; \mathcal{D}]$ where $\pi^\iota, \pi^o : \mathsf{Name} \xrightarrow{fin} \mathsf{Type}$ are the *input* and *output signature,* respectively, and $\mathcal{D}$ is a binary relation on Name called the *dependency* relation. The first two components are standard for module calculi (see [5]), while $\mathcal{D}$ keeps track of the input names an output name depends on, and will be used later in typing rules for executable configurations. Hence, typing rules for non-executable

$$(M\text{-basic})\ \frac{\begin{array}{c}\{\Gamma^{\iota}, \Gamma^{\rho} \vdash_e o(X) : \pi^{o}(X) \mid X \in \mathsf{dom}(o)\} \\ \{\Gamma^{\iota}, \Gamma^{\rho} \vdash_e \rho(x) : \Gamma^{\rho}(x) \mid x \in \mathsf{dom}(\rho)\}\end{array}}{\vdash_M [\iota;\ o;\ \rho] : \left[\pi^{\iota};\ \pi^{o};\ \mathcal{D}^{[\iota;\, o;\, \rho]}\right]} \quad \begin{array}{l}\mathsf{dom}(\pi^{\iota}) = \mathsf{img}(\iota) \\ \mathsf{dom}(\pi^{o}) = \mathsf{dom}(o) \\ \Gamma^{\iota} = \pi^{\iota} \circ \iota \\ \mathsf{dom}(\Gamma^{\rho}) = \mathsf{dom}(\rho)\end{array}$$

$$(M\text{-sum})\ \frac{\begin{array}{c}\vdash_M M_1 : [\pi^{\iota}{}_1;\ \pi^{o}{}_1;\ \mathcal{D}_1] \\ \vdash_M M_2 : [\pi^{\iota}{}_2;\ \pi^{o}{}_2;\ \mathcal{D}_2]\end{array}}{\vdash_M M_1 + M_2 : [\pi^{\iota}{}_1 \cup \pi^{\iota}{}_2;\ \pi^{o}{}_1, \pi^{o}{}_2;\ \mathcal{D}_1 \cup \mathcal{D}_2]}$$

$$(M\text{-link})\ \frac{\vdash_M M : [\pi^{\iota}{}_1, \pi^{\iota}{}_2;\ \pi^{o};\ \mathcal{D}]}{\vdash_M \mathsf{link}_{\sigma} M : [\pi^{\iota}{}_2;\ \pi^{o};\ \mathsf{link}_{\sigma}\mathcal{D}]} \quad \sigma : \pi^{\iota}{}_1 \to \pi^{o}$$

$$(M\text{-reduct})\ \frac{\vdash_M M : [\pi^{\iota};\ \pi^{o};\ \mathcal{D}]}{\vdash_M {}_{\sigma^{\iota}}|M|_{\sigma^{o}} : \left[\pi'^{\iota};\ \pi'^{o};\ {}_{\sigma^{\iota}}|\mathcal{D}|_{\sigma^{o}}\right]} \quad \begin{array}{l}\sigma^{\iota} : \pi^{\iota} \to \pi'^{\iota} \\ \sigma^{o} : \pi'^{o} \to \pi^{o}\end{array}$$

Figure 4. Typing rules for module expressions

configurations exactly correspond to typing rules for module expressions in *CMS* except for calculation of dependencies.

The definition of $\mathcal{D}$ as well as the corresponding operators on it defined in the sequel has been inspired by the CMS_v calculus [12]. Note, however, that here we have preferred to consider the inverse relation and that we do not need to deal with labelled multi-graphs. Intuitively, if $(Y, X) \in \mathcal{D}$ then *Y* depends on *X*, that is, *Y* is an output component of *M* associated with a core expression which (either directly or indirectly) refers to a deferred variable x which is mapped to the input component *X*. If $\mathcal{D}$ is a dependency relation, then we will write $Y \overset{\mathcal{D}}{\to} X$ for $(Y, X) \in \mathcal{D}$.

In rule (*M*-basic), we denote by $\Gamma \vdash_e e : \tau$ the typing judgment for core expressions, meaning that e is a well-formed expression of type τ in Γ, where Γ is a mapping from variables to core types. Moreover, $\mathcal{D}^{[\iota;\, o;\, \rho]}$ denotes the dependency relation induced by a basic module $[\iota;\ o;\ \rho]$, defined as follows. For $y \in \mathsf{dom}(\rho), x \in \mathsf{dom}(\iota) \cup \mathsf{dom}(\rho)$, let us write $y \to_{\rho} x$ iff $x \in \mathsf{FV}(\rho(y))$, and denote by $\to^{*}_{\rho}$ the transitive and reflexive closure of $\to_{\rho}$. Then, for all $Y \in \mathsf{dom}(o)$, $X \in \mathsf{img}(\iota)$, $Y \overset{\mathcal{D}^{[\iota;\, o;\, \rho]}}{\to} X$, iff there exist $y \in \mathsf{FV}(o(Y)), x \in \mathsf{dom}(\iota)$ s.t. $y \to^{*}_{\rho} x$ and $\iota(x) = X$.

The (*M*-sum) typing rule allows sharing of input components having the same name and type, while preventing output components from being shared. Recall that $f_1 \cup f_2$ denotes the union of two compatible partial functions, while f_1, f_2 denotes the union of two partial functions with disjoint domain. The dependency relation is the union of the dependency relations of the arguments.

In the (*M*-link) typing rule, the side-condition having the form $\sigma : \pi_1 \to \pi_2$ ensures that the renaming σ preserves types; formally, this means that $\sigma : \mathsf{dom}(\pi_1) \to \mathsf{dom}(\pi_2)$ and $\sigma(X) = Y \Rightarrow \pi_1(X) = \pi_2(Y)$. The dependency relation $\mathsf{link}_{\sigma}\mathcal{D}$ is defined as follows:

$\mathsf{link}_{\sigma}\mathcal{D} \triangleq (\mathcal{D} \cup \mathcal{D}_{\mathsf{add}}{}^{*}) \setminus \mathcal{D}_{\mathsf{remove}}$, where

$$\mathcal{D}_{\mathsf{add}} \triangleq \left\{(Y, X), (X, \sigma(X)), (\sigma(X), Z) \mid Y \overset{\mathcal{D}}{\to} X \wedge \sigma(X) \overset{\mathcal{D}}{\to} Z\right\}$$

$$\mathcal{D}_{\mathsf{remove}} \triangleq \{(Y, X), (X, \sigma(X)) \mid Y \overset{\mathcal{D}_{\mathsf{add}}^{*}}{\to} X \wedge X \in \mathsf{dom}(\sigma)\}$$

The intuition behind this definition is the following: any output name *Y* depending on an input name *X* that is going to be linked (that is, $X \in \mathsf{dom}(\sigma)$) will use the definition of the output component $\sigma(X)$ to which *X* is linked, which in turn may depend on

$$(C\text{-basic})\ \frac{\begin{array}{c}\{\Gamma^\iota,\Gamma^\rho\vdash_e o(X):\pi^o(X)\mid X\in\mathsf{dom}(o)\}\\ \{\Gamma^\iota,\Gamma^\rho\vdash_e \rho(x):\Gamma^\rho(x)\mid x\in\mathsf{dom}(\rho)\}\\ \Gamma^\iota,\Gamma^\rho\vdash_e e:\tau\end{array}}{\vdash_C <[\iota;\,o;\,\rho]\,,\ e>:\left(\left[\pi^\iota;\,\pi^o;\,\mathcal{D}^{[\iota;o;\rho]}\right],\mathcal{N}^{<[\iota;o;\rho],\,e>}\Rightarrow\tau\right)}\ \begin{array}{l}\mathsf{dom}(\pi^\iota)=\mathsf{img}(\iota)\\ \mathsf{dom}(\pi^o)=\mathsf{dom}(o)\\ \Gamma^\iota=\pi^\iota\circ\iota\\ \mathsf{dom}(\Gamma^\rho)=\mathsf{dom}(\rho)\end{array}$$

$$(C\text{-sum})\ \frac{\begin{array}{c}\vdash_C C:([\pi^\iota{}_C;\,\pi^o{}_C;\,\mathcal{D}_C]\,,\mathcal{N}\Rightarrow\tau)\\ \vdash_M M:[\pi^\iota{}_M;\,\pi^o{}_M;\,\mathcal{D}_M]\end{array}}{\vdash_C C+M:([\pi^\iota{}_C\cup\pi^\iota{}_M;\,\pi^o{}_C,\pi^o{}_M;\,\mathcal{D}_C\cup\mathcal{D}_M]\,,\mathcal{N}\Rightarrow\tau)}$$

$$(C\text{-link})\ \frac{\vdash_C C:([\pi^\iota{}_1,\pi^\iota{}_2;\,\pi^o;\,\mathcal{D}]\,,\mathcal{N}\Rightarrow\tau)}{\vdash_C \mathsf{link}_\sigma C:\left([\pi^\iota{}_2;\,\pi^o;\,\mathsf{link}_\sigma\mathcal{D}]\,,\mathsf{link}_\sigma^{\mathcal{D}}\mathcal{N}\Rightarrow\tau\right)}\ \sigma:\pi^\iota{}_1\rightarrow\pi^o$$

$$(C\text{-reduct})\ \frac{\vdash_C C:([\pi^\iota;\,\pi^o;\,\mathcal{D}]\,,\mathcal{N}\Rightarrow\tau)}{\vdash_C {}_{\sigma^\iota}|C|_{\sigma^o}:\left(\left[\pi'^\iota;\,\pi'^o;\,{}_{\sigma^\iota}|\mathcal{D}|_{\sigma^o}\right],\sigma^\iota(\mathcal{N})\Rightarrow\tau\right)}\ \begin{array}{l}\sigma^\iota:\pi^\iota\rightarrow\pi'^\iota\\ \sigma^o:\pi'^o\rightarrow\pi^o\end{array}$$

$$(\text{sel})\ \frac{\vdash_M M:[\pi^\iota;\,\pi^o;\,\mathcal{D}]}{\vdash_C M.X:([\pi^\iota;\,\pi^o;\,\mathcal{D}]\,,\mathcal{N}\Rightarrow\pi^o(X))}\ \mathcal{N}=\{Y\mid X\xrightarrow{\mathcal{D}}Y\}$$

Figure 5. Typing rules for configurations

some input name Z; hence, in the linking process the new dependencies obtained by computing the transitive closure of $\mathcal{D}_{\mathsf{add}}$ (denoted by $\mathcal{D}_{\mathsf{add}}{}^*$) must be added. Then all the dependencies involving the linked names $X\in\mathsf{dom}(\sigma)$ are removed.
In the (M-reduct) typing rule, the dependency relation ${}_{\sigma^\iota}|\mathcal{D}|_{\sigma^o}$ is defined as follows:
${}_{\sigma^\iota}|\mathcal{D}|_{\sigma^o}\triangleq\{(Y,\sigma^\iota(X))\mid\sigma^o(Y)\xrightarrow{\mathcal{D}}X\}$
Note that in the side-condition we again use the notation introduced in (M-link) to ensure that renamings preserve types.
The typing judgment for executable configurations has the form $\vdash_C C:([\pi^\iota;\,\pi^o;\,\mathcal{D}]\,,\mathcal{N}\Rightarrow\tau)$, meaning that C is a well-formed executable configuration of type $([\pi^\iota;\,\pi^o;\,\mathcal{D}]\,,\mathcal{N}\Rightarrow\tau)$. The first component $[\pi^\iota;\,\pi^o;\,\mathcal{D}]$ has the same meaning as for module expressions, whereas $\mathcal{N}\Rightarrow\tau$, with $\mathcal{N}$ set of names and τ core type, means that the expression to be executed in the configuration has type τ providing that all (input) names in $\mathcal{N}$ can be eventually linked (with the proper type).
In rule (C-basic), $\mathcal{N}^{<[\iota;o;\rho],\,e>}$ denotes the set of input names the expression e depends on, defined as follows:
$\mathcal{N}^{<[\iota;o;\rho],\,e>}\triangleq\{X\mid y\rightarrow^*_\rho x,\iota(x)=X,y\in\mathsf{FV}(e)\}$
Since the sum operator just glues modules together without linking any input name, in rule (C-sum) the second component $\mathcal{N}\Rightarrow\tau$ remains unchanged.
In the (C-link) typing rule, the set of names $\mathsf{link}_\sigma^{\mathcal{D}}\mathcal{N}$ is defined as follows:
$\mathsf{link}_\sigma^{\mathcal{D}}\mathcal{N}\triangleq(\mathcal{N}\cup\mathcal{N}^{\mathcal{D}}_{\mathsf{sum}})\setminus\mathsf{dom}(\sigma)$, where $\mathcal{N}^{\mathcal{D}}_{\mathsf{sum}}=\{Z\mid X\in\mathcal{N}\wedge\sigma(X)\overset{\mathsf{link}_\sigma\mathcal{D}}{\rightarrow}Z\}$.
Indeed, before removing from $\mathcal{N}$ all linked input names in $\mathsf{dom}(\sigma)$, all new dependencies reachable from $\mathcal{N}$ with respect to the relation $\mathsf{link}_\sigma\mathcal{D}$ must be added.
In the (C-reduct) typing rule, $\sigma^\iota(\mathcal{N})$ denotes the set $\{\sigma^\iota(X)\mid X\in\mathcal{N}\}$.
Finally, in (C-sel) the set $\mathcal{N}$ corresponds to all the input names the output component X depends on, whereas the type τ of the expression to be executed coincides with the type of X.

5 Results

In this section we collect all the technical results about the calculus. In particular, we state the Church Rosser property for the reduction relation and the Subject reduction and Progress properties. Clearly, these results hold providing that the corresponding properties are verified at the core level as well. Moreover, we also assume the core language to be such that: if $e \xrightarrow[e]{} e'$, then $\mathsf{FV}(e') \subseteq \mathsf{FV}(e)$; if $\Gamma \vdash_e \mathcal{E}[e] : \tau$, then there exists τ' such that $\Gamma \vdash_e e' : \tau'$ and for all e' such that $\Gamma \vdash_e e' : \tau'$ we have that $\Gamma \vdash_e \mathcal{E}[e'] : \tau$.

PROPOSITION 3 *The reduction relation* $\longrightarrow$ *is confluent.*

THEOREM 4 (SUBJECT REDUCTION) *If* $\vdash_M M : [\pi^\iota; \pi^o; \mathcal{D}]$ *and* $M \longrightarrow M'$, *then* $\vdash_M M' : [\pi^\iota; \pi^o; \mathcal{D}]$. *If* $\vdash_C C : ([\pi^\iota; \pi^o; \mathcal{D}], \mathcal{N} \Rightarrow \tau)$ *and* $C \longrightarrow C'$, *then* $\vdash_C C' : ([\pi^\iota; \pi^o; \mathcal{D}], \mathcal{N} \Rightarrow \tau)$.

To state the Progress property, we have to define the set VConf of values for the terms of the calculus:

$$\begin{array}{lll} CV \in \mathsf{VConf} & ::= & <[\iota; o; \rho], ev> \mid CV + MV \mid \mathsf{link}_\sigma CV \mid {}_{\sigma^\iota}|CV|_{\sigma^o} \\ MV \in \mathsf{VMExp} & ::= & [\iota; o; \rho] \end{array}$$

where with ev we denote a value at the core level.

THEOREM 5 (PROGRESS)
If $\vdash_M M : [\pi^\iota; \pi^o; \mathcal{D}]$ *and* $M \notin \mathsf{VMExp}$, *then there exists* M' *s.t.* $M \longrightarrow M'$. *If* $\vdash_C C : ([\pi^\iota; \pi^o; \mathcal{D}], \emptyset \Rightarrow \tau)$ *and* $C \notin \mathsf{VConf}$, *then there exists* C' *s.t.* $C \longrightarrow C'$.

Note that progress for executable configurations holds only if the expression to be executed does not depend on any input name.

6 Conclusion

We have defined CMS^ℓ, an extension of *CMS* [5] where operators on modules are performed on demand, when needed by the execution of a program, rather than eagerly, before any access to module components. We have provided a sound type system for the calculus, relying on a dependency analysis which ensures that execution never needs to access module components which cannot become available by performing reconfiguration steps.

The relevance of this work is twofold. On one hand, whereas lazy evaluation has been extensively studied in the context of variants of lambda-calculus (see, e.g., [6]), there was to our knowledge no previous attempt at analyzing this feature in the context of record-based calculi or module calculi. We believe that the combination of laziness with the computational paradigm based on record selection is a stimulating subject for research, which could provide new programming patterns and be used in a wide variety of contexts. In this respect, the contribution of this paper is to provide the first step in this direction.

On the other hand, a more specific motivation for CMS^ℓ is the need for foundational calculi providing an abstract framework for dynamic reconfiguration (that is, interleaving of reconfiguration steps and execution steps). Indeed, though the area of unanticipated software evolution continues attracting large interest, with its foundations studied in, e.g., [14], there is a little amount of work at to our knowledge going toward

the development of abstract models for dynamic reconfiguration, analogous to those which exist for the static case (where the configuration phase always precedes execution) [8, 15, 5]. Apart from the wide literature concerning concrete dynamic linking mechanisms in existing programming environments [9, 10], we mention [7], which presents a simple calculus modeling dynamic software updating, where modules are just records, many versions of the same module may coexist and update is modeled by an external transition which can be enforced by an update primitive in code, and [1], where dynamic linking is studied as the programming language counterpart to the axiom of choice. Finally, we have proposed in a recent paper [3] a calculus for dynamic linking (CDL) partly driven by the same objectives as CMS^ℓ, that is, to define a kernel calculus able to express some form of dynamic reconfiguration abstracting from the details of the particular underlying programming language. Here below we briefly compare the two proposals.
In *CDL*, we did not attempt at introducing dynamic features in a pure module calculus, but rather to combine a module calculus with explicit imperative features. Indeed, terms of *CDL* are *configurations* consisting of a *linkset* (corresponding to a module expression in the terminology used in this paper) and a command. Configurations can evolve in two ways: either by simplifying the linkset expression (that is, performing a reconfiguration step) or by performing a step in the execution of the command. In particular, classical module operators such as sum and (static) link must be performed before execution of the command starts; however, a new operator is introduced, called *dynamic link*, which is only performed on demand, after execution of the command has started. More precisely, a dynamic link operator for a component X (in *CDL* linking is performed on a per-name basis) is only performed if the execution needs a deferred variable, say x, which is associated to X.
Both *CDL* and CMS^ℓ proposals give, in our opinion, an important contribution toward the development of a framework for dynamic reconfiguration, but both have some (different) limitations. In *CDL*, only a limited form of interleaving between the reconfiguration and the execution phase is allowed, since the classical module operators, notably sum, must be performed before execution starts. Moreover, run-time reconfiguration ability is obtained by adding new ingredients (the dynamic link operator). In CMS^ℓ, on the contrary, no new operator is added to a standard module calculus, and there is true interleaving of the reconfiguration and execution phase, since no module operator needs to be performed before evaluating a module component. However, in CMS^ℓ this interleaving is handled by a fixed policy, in the sense that standard execution steps always take the precedence over reconfiguration steps, unless they are needed since execution would otherwise get stuck. Moreover, all reconfiguration steps are planned statically.
We believe that both CMS^ℓ and *CDL* can be seen as early steps towards more powerful calculi able to handle interleaving of reconfiguration and standard execution steps in a liberal way and to encode all the possibilities mentioned above. An important issue to be investigated in parallel is the expressive power of such calculi, by showing which kind of real-world reconfiguration mechanisms can be modeled and which kind cannot by each of them. Though the practical motivations of calculi for dynamic reconfiguration are certainly founded, a more detailed analysis of this connection is at a very initial stage, due to the youth of the trend toward such models itself. We have presented a preliminary attempt in [11], where we have used a particular instantiation

of *CDL* to encode a toy language, called JL, which provides an abstract view of the mechanism of dynamic class loading with multiple loaders as in Java.

Acknowledgments

We warmly thank Eugenio Moggi, Joe Wells, Henning Makholm and Sebastien Carlier for useful comments on previous drafts of this work.

References

[1] Martin Abadi, Goerges Gonthier, and Benjamin Werner. Choice in dynamic linking. In *FOSSACS'04 - Foundations of Software Science and Computation Structures 2004,* Lecture Notes in Computer Science. Springer, 2004.

[2] D. Ancona, S. Fagorzi, E. Moggi, and E. Zucca. Mixin modules and computational effects. In Jos C. M. Baeten et al., editors, *International Colloquium on Automata, Languages and Programming 2003,* number 2719 in Lecture Notes in Computer Science, pages 224–238. Springer, 2003.

[3] D. Ancona, S. Fagorzi, and E. Zucca. A calculus for dynamic linking. In C. Blundo and C. Laneve, editors, *Italian Conf. on Theoretical Computer Science 2003,* number 2841 in Lecture Notes in Computer Science, pages 284–301, 2003.

[4] D. Ancona and E. Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science,* 8(4):401–446, August 1998.

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

[6] Z. M. Ariola and M.Felleisen. The call-by-need lambda calculus. *Journ. of Functional Programming,* 7(3):265-301, 1997.

[7] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating (Extended Abstract). In *USE'03 - Workshop on Unexpected Software Evolution,* 2003.

[8] L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997,* pages 266–277. ACM Press, 1997.

[9] S. Drossopoulou. Towards an abstract model of Java dynamic linking and verfication. In R. Harper, editor, *TIC'00 - Third Workshop on Types in Compilation (Selected Papers),* volume 2071 of *Lecture Notes in Computer Science,* pages 53–84. Springer, 2001.

[10] S. Drossopoulou, G. Lagorio, and S. Eisenbach. Flexible models for dynamic linking. In Pierpaolo Degano, editor, *ESOP 2003 - European Symposium on Programming 2003,* pages 38–53, April 2003.

[11] S. Fagorzi, E. Zucca, and D. Ancona. Modeling multiple class loaders by a calculus for dynamic linking. In *ACM Symp. on Applied Computing (SAC 2004), Special Track on Object-Oriented Programming Languages and Systems.* ACM Press, 2004. To appear.

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

[13] X. Leroy. A modular module system. *Journal of Functional Programming,* 10(3):269–303, May 2000.

[14] Tom Mens and Guenther Kniesel. Workshop on foundations of unanticipated software evolution. ETAPS 2004, http://joint.org/fuse2004/, 2004.

[15] J.B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. In *ESOP 2000 - European Symposium on Programming 2000,* number 1782 in Lecture Notes in Computer Science, pages 412–428. Springer, 2000.