Mixin modules for dynamic rebinding[⋆]

Davide Ancona, Sonia Fagorzi, and Elena Zucca

DISI - Università di Genova
Via Dodecaneso, 35, 16146 Genova (Italy)
`email: {davide,fagorzi,zucca}@disi.unige.it`

Abstract. *Dynamic rebinding* is the ability of changing the definitions of names at execution time. While dynamic rebinding is clearly useful in practice, and increasingly needed in modern systems, most programming languages provide only limited and ad-hoc mechanisms, and no adequate semantic understanding currently exists.
Here, we provide a simple and powerful mechanism for dynamic rebinding by means of a calculus $\mathit{CMS}^{\ell,v}$ of *mixin modules* (mutually recursive modules allowing redefinition of components) where, differently from the traditional approach, module operations can be performed after selecting and executing a module component: in this way, execution can refer to *virtual* components, which can be rebound when module operators are executed. In particular, in our calculus module operations are performed on demand, when execution would otherwise get stuck.
We provide a sound type system, which ensures that execution never tries to access module components which cannot become available, and show how the calculus can be used to encode a variety of real-world dynamic rebinding mechanisms.

1 Introduction

In the last years considerable effort has been invested in developing kernel module/fragment calculi [12, 7, 23, 21, 7, 19] providing foundations for flexible manipulation and combination of software components. In particular, a simple unifying notion emerged from this research stream is that of *mixin module* [11, 3], that is, a module which allows late (re)definition of components. In a mixin module components are either defined inside the module (exported) or *deferred* (imported), that is, to be provided later by means of combination with other modules (notably, in a mutually recursive way by a symmetric sum operator). Moreover, some defined components can be *virtual*, that is, can be later modified as an effect of combination with other modules (notably, by an overriding operator), so that all their internal references are dynamically rebound to the new definition. The possibility of defining virtual components is a generalization to an arbitrary context of software composition of a key idea of the object-oriented

[⋆] Partially supported by Dynamic Assembly, Reconfiguration and Type-checking - EC project IST-2001-33477, APPSEM II - Thematic network IST-2001-38957i, and MIUR EOS - Extensible Object Systems.

approach, that is, the ability of writing code fragments (classes in this case) where components (methods) are simultaneously ready to be used and available to be modified (*open-closed* property).
Calculi supporting mixin modules, such as the Calculus of Module Systems (shortly *CMS*) developed by two of the authors[7], can be used to encode and compare on a formal basis a large variety of existing mechanisms for software composition, including parameterized modules like ML functors, overriding, extra-linguistic mechanisms like those provided by a linker. However, these calculi are based on a *static* view of software manipulation, hence fail in many ways to be adequate to model modern software systems, which become increasingly dynamic. For instance, programming environments such as those of Java and C# support dynamic linking, and we can expect in the future more and more forms of *reconfiguration* interleaved with standard *execution* steps; when values of computations are marshaled from a running program and moved elsewhere, some of their identifiers may need to be dynamically rebound; systems which provide uninterrupted service must be dynamically updated.
All these situations could be hardly represented in, e.g., *CMS*, even though the notion of virtual component, allowing the same name to be bound to different definitions during successive steps of configuration of a software system, seems to exactly correspond to rebinding. This is due to the fact that in *CMS* and similar calculi all module operators must be performed *before* starting execution of a program, that is, evaluation of a module component. Hence, virtual components can be usefully employed to rebind the same name to different definitions, and thus reuse in different ways the same module in different contexts, but this rebinding is *static* in the sense that only closed modules (that is, with no deferred or virtual components) can be actually used at execution time.
Here, we are able to obtain a simple and powerful calculus for dynamic rebinding from *CMS* by developing the following simple key ideas.

- Components of open modules can be selected and executed, keeping their module context. In this way, execution of module operators can be interleaved with *program execution*, that is, execution of a module component in the context of the components offered by the module. We already introduced this idea in CMS^{ℓ} [6], where in particular we adopted a lazy strategy which performs reconfiguration steps (execution of module operators) only if necessary, that is, when program execution would otherwise gets stuck (since a not yet available component is needed.)
- Program execution refers to not only *local*, but also *virtual* components, that is, components which are associated with a definition which is directly available to the executing program and can also be redefined by performing module operators. This conceptually simple extension greatly enhances the expressive power. Indeed, in CMS^{ℓ}, reconfiguration steps can either be performed or not depending on which components program execution needs, but when a component is bound to a definition this binding can no longer be changed. On the contrary, in $\mathit{CMS}^{\ell,v}$ execution can refer to components which can be redefined when module operators are executed.

Another important novelty w.r.t. CMS^{ℓ} is that $CMS^{\ell,v}$ keeps the full expressive power of higher-order features of *CMS*. This allows to express interaction of execution at different levels (e.g., modules with module components, triggering of a local module simplification inside program execution, and so on).
In Section 2 and Section 3 we formally define the calculus. In Section 4 we show how the calculus can be used to model real-world dynamic rebinding requirements. In Section 5 we provide a sound type system, which ensures that execution will never try to access module components which cannot become available. Section 6 collects the technical results (limited to the claims), and finally Section 7 contains concluding remarks and directions for further work.

2 Syntax

Notations We denote by $A \overset{fin}{\to} B$ the set of partial functions f from A into $B = \mathsf{cod} f$ with finite domain $\mathsf{dom}(f) \subseteq A$. For I set of indexes, $a_i \in A$, $b_i \in B$, for $i \in I$, we denote by ${a_i : b_i}^{i\in I}$ the partial function f s.t. $\mathsf{dom}(f) = \{a_i \mid i \in I\}$, $f(a_i) = b_i$ for $i \in I$. We will use the following operators on partial functions: f, g is the union of two functions with disjoint domain; $f|g$ means that f, g are *compatible*, that is, s.t. $f(a) = g(a)$ for all $a \in \mathsf{dom}(f) \cap \mathsf{dom}(g)$; $f \cup g$ is the union of two compatible functions; $\circ$ is the composition of functions; $f\backslash_A$ is the restriction of f to the domain $\mathsf{dom}(f) \setminus A$; we write $f\backslash_a$ instead of $f\backslash_{\{a\}}$.

The syntax of $CMS^{\ell,v}$ is given in Fig.1.

$e \in \mathsf{Exp}$	$::=$		**expression**
		$\ldots$	core operators
	$\mid$	x	variable
	$\mid$	$[\iota; o; \rho]$ $(\mathsf{dom}(\iota)\cap\mathsf{dom}(\rho)=\emptyset)$	basic module
	$\mid$	$[\iota; o; \rho \mid e]$ $(\mathsf{dom}(\iota)\cap\mathsf{dom}(\rho)=\emptyset)$	basic configuration
	$\mid$	$e_1 + e_2$	sum
	$\mid$	$\mathsf{freeze}_X e$	freeze
	$\mid$	$e\backslash_X$	delete
	$\mid$	$e\downarrow_X$	run
	$\mid$	$e\uparrow$	result
$\iota : \mathsf{Var} \overset{fin}{\to} \mathsf{Name}$			**input** assignment
$o : \mathsf{Name} \overset{fin}{\to} \mathsf{Exp}$			**output** assignment
$\rho : \mathsf{Var} \overset{fin}{\to} \mathsf{Exp}$			**local** assignment

Fig. 1. Syntax

We assume an infinite set Name of *names* X, and an infinite set Var of *variables* x. Names are used to refer to a module from the outside (hence they are used by module operators), while variables are used to refer to a (basic) module from a program executing in the context of the components offered by this

module. 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., [7] for an extended discussion of this point).
As *CMS* and CMS^{ℓ}, $\mathit{CMS}^{\ell,v}$ is a parametric and stratified calculus, which can be instantiated over different core calculi. However, while in *CMS* [7] this dependence on the core level is represented in a more rigorous way by using explicit substitutions, here we adopt for simplicity a less formal approach where we assume module expressions to be merged with expressions of the core calculus (that is, the dots in the syntax correspond to core productions). In the examples we assume that core expressions include integers with the usual operations.
Basic modules are as in *CMS* and consist of three parts: the *input assignment* ι, which is a mapping from variables into *input names*, the *output assignment* o, which is a mapping from *output names* into expressions, and the *local assignment* ρ, which is a mapping from *local variables* into expressions. Input names are called *virtual* if they are output names as well, *deferred* otherwise; variables in the domain of ι are called either virtual or deferred depending on the associated name. A basic configuration is a pair $[\iota; o; \rho \mid e]$, consisting of a basic module and an expression, called *program*.
Both basic modules and basic configurations are well-formed only if the sets of deferred and virtual variables and that of local variables are disjoint.
Operators *sum*, *freeze* and *delete* are a simplified version of *CMS* module operators, and provide primitive ways to manipulate and combine software fragments. *Modules* can be constructed by applying these operators on top of basic modules, and *configurations* can be constructed by applying these operators on top of basic modules and at least one basic configuration (actually, exactly one in well-behaving terms, see in the sequel, hence we can correctly talk of *the* program running inside a configuration). Operator $\_\!\downarrow\!\_$ allows to obtain a basic configuration from a (basic) module, by starting the execution of a module component. Operator $\_\uparrow$ extracts from a configuration the (final result of) the program. Operators will be explained more in detail when introducing reduction rules.

3 Semantics

In this section we give the semantics of the calculus. Reduction rules are given in Fig. 2 and Fig. 3.
By definition, the one step reduction relation $\longrightarrow$ is the relation over well-formed terms inductively defined by the rules. For sake of clarity, we write also some side conditions which are redundant since implied by the fact that terms must be well-formed.

The semantics is given by using evaluation contexts (to control the evaluation order) and redexes (reducible expressions), following the approach of Felleisen and Friedman [18]. Rule ($\mathcal{E}$) is the usual contextual closure, where evaluation contexts $\mathcal{E}$ include a non specified set of core evaluation contexts, and the metavariable r

Evaluation contexts

$$\begin{array}{l}\mathcal{E} ::= \square \mid \ldots \mid [\iota; o; \rho \mid \mathcal{E}] \mid \mathcal{E} + e \mid \mathcal{E} \setminus_X \mid \mathsf{freeze}_X \mathcal{E} \mid \mathcal{E}{\downarrow_X} \mid \mathcal{E}{\uparrow} \\ \quad \mid \; [\iota; o; \rho] + \mathcal{E} \mid [\iota; o; \rho \mid \mathcal{E}[x]] + \mathcal{E} \;\; (x \in \mathsf{dom}(\iota) \wedge \iota(x) \notin \mathsf{dom}(o))\end{array}$$

Contextual closure and core execution

$$(\mathcal{E}) \; \frac{r \longrightarrow e}{\mathcal{E}[r] \longrightarrow \mathcal{E}[e]}$$

. . . (rules for core operators)

Module simplification

$$\text{(m-sum)} \; \frac{}{\begin{array}{r}[\iota_1; o_1; \rho_1] + [\iota_2; o_2; \rho_2] \longrightarrow \\ [\iota_1, \iota_2; o_1, o_2; \rho_1, \rho_2]\end{array}} \; \begin{array}{l}\mathsf{dom}(\iota_1, \rho_1) \cap \mathsf{FV}\,([\iota_2; o_2; \rho_2]) = \emptyset \\ \mathsf{dom}(\iota_2, \rho_2) \cap \mathsf{FV}\,([\iota_1; o_1; \rho_1]) = \emptyset \\ \mathsf{dom}(\iota_1, \rho_1) \cap \mathsf{dom}(\iota_2, \rho_2) = \emptyset \\ \mathsf{dom}(o_1) \cap \mathsf{dom}(o_2) = \emptyset\end{array}$$

$$\text{(m-freeze)} \; \frac{}{\mathsf{freeze}_X [\iota; o; \rho] \longrightarrow \left[\iota\setminus_F; o; \rho, x : o(X)^{x \in F}\right]} \; \begin{array}{l}F = \{x \mid \iota(x) = X\} \\ F \neq \emptyset \Rightarrow X \in \mathsf{dom}(o)\end{array}$$

$$\text{(m-del)} \; \frac{}{[\iota; o; \rho] \setminus_X \longrightarrow [\iota; o\setminus_X; \rho]} \; X \in \mathsf{dom}(o)$$

Variable resolution and reconfiguration

$$\text{(local)} \; \frac{}{[\iota; o; \rho \mid \mathcal{E}[x]] \longrightarrow [\iota; o; \rho \mid \mathcal{E}\{\rho(x)\}]} \; \begin{array}{l}x \notin \mathsf{HB}\,(\mathcal{E}) \\ x \in \mathsf{dom}(\rho)\end{array}$$

$$\text{(virtual)} \; \frac{}{[\iota; o; \rho \mid \mathcal{E}[x]] \longrightarrow [\iota; o; \rho \mid \mathcal{E}\{o(\iota(x))\}]} \; \begin{array}{l}x \notin \mathsf{HB}\,(\mathcal{E}) \\ x \in \mathsf{dom}(\iota) \wedge \iota(x) \in \mathsf{dom}(o)\end{array}$$

$$\text{(sum)} \; \frac{}{\begin{array}{r}[\iota_1; o_1; \rho_1 \mid \mathcal{E}[x]] + [\iota_2; o_2; \rho_2] \longrightarrow \\ [\iota_1, \iota_2; o_1, o_2; \rho_1, \rho_2 \mid \mathcal{E}[x]]\end{array}} \; \begin{array}{l}x \notin \mathsf{HB}\,(\mathcal{E}) \\ x \in \mathsf{dom}(\iota_1) \wedge \iota_1(x) \notin \mathsf{dom}(o_1) \\ \mathsf{dom}(\iota_1, \rho_1) \cap \mathsf{FV}\,([\iota_2; o_2; \rho_2]) = \emptyset \\ \mathsf{dom}(\iota_2, \rho_2) \cap \mathsf{FV}\,([\iota_1; o_1; \rho_1 \mid \mathcal{E}[x]]) = \emptyset \\ \mathsf{dom}(\iota_1, \rho_1) \cap \mathsf{dom}(\iota_2, \rho_2) = \emptyset \\ \mathsf{dom}(o_1) \cap \mathsf{dom}(o_2) = \emptyset\end{array}$$

$$\text{(freeze)} \; \frac{}{\mathsf{freeze}_X [\iota; o; \rho \mid \mathcal{E}[x]] \longrightarrow \left[\iota\setminus_F; o; \rho, x : o(X)^{x \in F} \mid \mathcal{E}[x]\right]} \; \begin{array}{l}x \notin \mathsf{HB}\,(\mathcal{E}) \\ x \in \mathsf{dom}(\iota) \wedge \iota(x) \notin \mathsf{dom}(o) \\ F = \{x \mid \iota(x) = X\} \\ F \neq \emptyset \Rightarrow X \in \mathsf{dom}(o)\end{array}$$

$$\text{(del)} \; \frac{}{[\iota; o; \rho \mid \mathcal{E}[x]] \setminus_X \longrightarrow [\iota; o\setminus_X; \rho \mid \mathcal{E}[x]]} \; \begin{array}{l}x \notin \mathsf{HB}\,(\mathcal{E}) \wedge x \in \mathsf{dom}(\iota) \wedge \iota(x) \notin \mathsf{dom}(o) \\ X \in \mathsf{dom}(o)\end{array}$$

Fig. 2. Reduction rules

$e \in \mathsf{Exp}$	$\mathsf{FV}\,(e)$
$\ldots$	free variables for core operators
x	$\{x\}$
$[\iota; o; \rho]$	$\mathsf{FV}\,(o) \cup \mathsf{FV}\,(\rho) \setminus \mathsf{dom}(\iota, \rho)$
$[\iota; o; \rho \mid e]$	$(\mathsf{FV}\,(o) \cup \mathsf{FV}\,(\rho) \cup \mathsf{FV}\,(e)) \setminus \mathsf{dom}(\iota, \rho)$
$e_1 + e_2$	$\mathsf{FV}\,(e_1) \cup \mathsf{FV}\,(e_2)$
$e \setminus_X \mid \mathsf{freeze}_X e \mid e{\downarrow_X} \mid e{\uparrow}$	$\mathsf{FV}\,(e)$
$f : A \overset{fin}{\to} \mathsf{Exp}$	$\cup\{\mathsf{FV}\,(f(a)) \mid a \in \mathsf{dom}(f)\}$
$\mathcal{E}$	$\mathsf{HB}\,(\mathcal{E})$
$\Box$	$\emptyset$
$[\iota; o; \rho \mid \mathcal{E}]$	$\mathsf{dom}(\iota, \rho) \cup \mathsf{HB}\,(\mathcal{E})$
$\mathcal{E} + e \mid \mathcal{E} \setminus_X \mid \mathsf{freeze}_X \mathcal{E} \mid \mathcal{E}{\downarrow_X} \mid \mathcal{E}{\uparrow}$	$\mathsf{HB}\,(\mathcal{E})$

Table 1. Free variables and hole binders

ranges over redexes, that is, the left-hand sides of the consequence in (instantiations of) the other rules, called *computational*. We denote by $\mathcal{E}[e]$ the expression obtained by replacing by e the hole in context $\mathcal{E}$. *Reconfiguration contexts* $\mathcal{R}$ are special contexts used in rule (res-extract) and (res-var), as explained below.
The evaluation context $[x : X, \iota; o; \rho \mid \mathcal{E}[x]] + \mathcal{E}$ expresses the fact that in the sum of a configuration with a module the evaluation of the right-hand-side argument is only triggered when the configuration is fully reduced and the running program still needs reconfiguration steps to proceed (indeed, in this case the module needs to be reduced in order sum to be performed.)
We assume that computational rules for the core operators are provided.

Module simplification Simplification rules for sum, freeze and delete on modules are exactly as in *CMS*. We give here a brief description, referring to [7] for more detailed comments.

Sum The sum operation has the effect of gluing together two modules. The first two side conditions avoid undesired captures of free variables. Free variables of expressions are defined in Table 1, assuming that their definitions on core terms are provided. 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 (second side condition). These side conditions 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 (last side condition)[1]; however, in this case the reduction gets stuck since this conflict cannot be resolved by an α-conversion.

Freeze The freeze operator removes the name X appearing as index from the input names. All the virtual variables mapped by ι into it are *frozen*, that

[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.

is, become local, and take as defining expression the current definition of X in the output assignment. Hence, this definition must exist in case there is at least one variable mapped into X (side-condition); otherwise, the freeze operator has simply no effect. The name of the operator refers to the fact that, once a component has been frozen, other components will permanently refer to its current definition, even in case the component is updated from outside (by delete and then sum, see below).

Delete The delete operator removes an output name (which must be present in the module) with the associated definition.

Note that these three operators provide complementary capabilities for changing the status of a variable x in a basic module, as follows:

- A deferred variable can become virtual as an effect of the sum operator (if x is mapped by ι into an input name X, and X is an output name in the other argument of the sum).
- A virtual variable can become local as an effect of the freeze operator (if x is mapped into the name X appearing as index).
- A virtual variable can become deferred as an effect of the delete operator (if x is mapped into the name X appearing as index).

Local variables are not visible from outside a basic module (or basic configuration), hence cannot change their status.

Variable resolution and reconfiguration Rules (local) and (virtual) model the situation where program execution needs a variable which is either local or virtual, hence has a corresponding definition, in the enclosing basic configuration.
In both cases, program execution can proceed by replacing the variable by its defining expression.
Here and in the following rules, the side condition $x \notin \mathsf{HB}(\mathcal{E})$ expresses the fact that the occurrence of the variable x in the position denoted by the hole of the context $\mathcal{E}$ is free (that is, not captured by any binder around the hole). Hole binders are defined in Table 1 (we assume their definitions on core terms are provided). Finally, we denote by $\mathcal{E}\{e\}$ the capture avoiding substitution, with the expression e, of the hole $\Box$ in context $\mathcal{E}$.
These two rules, together with rules for core operators and contextual closure, model standard program execution (that is, execution which does not trigger reconfiguration steps), as illustrated by the following example.[2]

$$[x : X; X : 1; y : 2 \mid x + y] \xrightarrow{\text{(virtual)}} [x : X; X : 1; y : 2 \mid 1 + y] \xrightarrow{\text{(local)}}$$
$$[x : X; X : 1; y : 2 \mid 1 + 2] \xrightarrow{\text{(core)}} [x : X; X : 1; y : 2 \mid 3]$$

Note that, since a program can be in turn a configuration, variable resolution can take place at an outer configuration level:

[2] In examples we label reduction steps with the applied computational rule. We label with (core) reduction steps where we apply core computational rules.

$$[x:X;X:1;y:2 \mid [;;\mid x+y]] \xrightarrow{\text{(virtual)}} [x:X;X:1;y:2 \mid [;;\mid 1+y]] \xrightarrow{\text{(local)}}$$
$$[x:X;X:1;y:2 \mid [;;\mid 1+2]] \xrightarrow{\text{(core)}} [x:X;X:1;y:2 \mid [;;\mid 3]]$$

The side condition $x \notin \mathsf{HB}(\mathcal{E})$ ensures that a variable which is bound at an inner configuration level cannot be resolved at an outer level:

$$[x:X;X:1;y:2 \mid [x:Z;Z:5;\mid x+y]] \xrightarrow{\text{(virtual)}}$$
$$[x:X;X:1;y:2 \mid [x:X;X:5;\mid 5+y]]$$

$$[x:X;X:1;y:2 \mid [x:X;X:5;\mid x+y]] \not\longrightarrow$$
$$[x:X;X:1;y:2 \mid [x:X;X:5;\mid 1+y]]$$

where $\not\longrightarrow$ denotes a not allowed reduction step.
The fact that substitution is capture avoiding prevents variables from outer levels to be captured at an inner level:

$$[x:X;X:y+1;y:2 \mid [;;y:3 \mid x]] \xrightarrow{\text{(virtual)}} [x:X;X:z+1;z:2 \mid [;;y:3 \mid z+1]]$$
$$[x:X;X:y+1;y:2 \mid [;;y:3 \mid x]] \not\longrightarrow [x:X;X:y+1;y:2 \mid [;;y:3 \mid y+1]]$$

A different choice, allowing variables to "migrate" into inner levels, would correspond to a form of dynamic binding for variables.
The following three rules model the situation where program execution needs a variable which is deferred, that is, is bound in the current basic module but has no corresponding definition. In this case, a *reconfiguration* step is triggered: more precisely, the innermost enclosing module operator is performed.
As combined effect of the rules illustrated until now, execution proceeds by standard execution steps 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 virtual and rule (virtual) can be applied, as illustrated below.

$$[x:X;;y:2 \mid x+y] + [;X:1;] \xrightarrow{\text{(sum)}} [x:X;X:1;y:2 \mid x+y] \xrightarrow{\text{(virtual)}}$$
$$[x:X;X:1;y:2 \mid 1+2] \xrightarrow{\text{(core)}} [x:X;X:1;y:2 \mid 3]$$

As happens for variable resolution, also reconfiguration steps can take place at an outer configuration level if the needed variable is not bound yet.
Note that, whereas sum of two modules and sum of a configuration with a module (conventionally taken in this order) are handled by rule (m-sum) and (sum), respectively, there is no rule for sum of two configurations which, hence, gets stuck (and will be rejected by the type system). This corresponds to the fact that we are considering a sequential calculus, in which there is only one executing program at a given configuration level.

Values and reconfiguration contexts

$v \in \mathsf{Val} ::= \ldots \mid [\iota; o; \rho] \mid \mathcal{R}[\iota; o; \rho \mid v]$
$\mathcal{R} \quad ::= \Box \mid \mathcal{R} + e \mid \mathcal{R} \setminus_X \mid \mathsf{freeze}_X \mathcal{R}$

Run and result

$$\text{(run)} \quad \frac{}{[\iota; o; \rho]\downarrow_X \longrightarrow [\iota; o; \rho \mid o(X)]} \quad X \in \mathsf{dom}(o)$$

$$\text{(res-extract)} \quad \frac{}{(\mathcal{R}[\iota; o; \rho \mid v])\uparrow \longrightarrow v} \quad \mathsf{FV}\,(v) \cap (\mathsf{dom}(\iota, \rho)) = \emptyset$$

$$\text{(res-var)} \quad \frac{\mathcal{R}[\iota; o; \rho \mid x] \longrightarrow \mathcal{R}'[\iota'; o'; \rho' \mid e]}{(\mathcal{R}[\iota; o; \rho \mid v[x]])\uparrow \longrightarrow (\mathcal{R}'[\iota'; o'; \rho' \mid v\{e\}])\uparrow} \quad x \in \mathsf{dom}(\iota, \rho)$$

Fig. 3. Reduction rules (cont)

Run and result Rules in Fig.3 deal with introduction and elimination of a configuration level, respectively. In rule (run), the operator $\downarrow$ constructs an initial configuration by taking as program an output component of a basic module.

$$[x : X; X : 1, Z : x + y; y : 2]\downarrow_Z \overset{\text{(run)}}{\longrightarrow} [x : X; X : 1, Z : x + y; y : 2 \mid x + y] \overset{\text{(virtual)}}{\longrightarrow} \ldots$$

The following two rules deal with the operator $\uparrow$, which extracts the program from a configuration level. Formally, a configuration level for a program e is modeled by an expression of the form $\mathcal{R}[\iota; o; \rho \mid e]$ where $\mathcal{R}$ is a context consisting only of reconfiguration operators.
Rule (res-extract) allows to extract the program from a configuration level if it is a value which contains no variables bound at this level. Note that remaining reconfiguration operators are simply ignored, since they can no longer have any effect on the result of the computation. This is illustrated by the following example, where we assume to have lambda-abstractions in the core calculus.

$$[;;y : 2 \mid ([;;x : 1 \mid \lambda z.1 + y] + [;Z : 0;])\uparrow] \overset{\text{(res-extract)}}{\longrightarrow}$$
$$[;;y : 2 \mid \lambda z.1 + y]$$

If the program is a value still containing some variables bound at the current configuration level, these variables must be resolved before extracting the value. This is handled by rule (res-var), where we write $v[x]$ to denote a value which contains a free occurrence of x, and, analogously to the notation used for evaluation contexts, $v\{e\}$ to denote the expression obtained by replacing this occurrence by e. The effect we want to obtain is that the action needed to solve variable x is triggered (x is replaced by its definition if it is either local or virtual, and the innermost module operator in $\mathcal{R}$ is performed if x is deferred). For sake of brevity, we write just one compact rule instead of five rules analogous to those which handle resolution of a variable x in a program which is not a value (hence can be decomposed as $\mathcal{E}[x]$), that is, (local), (virtual), (sum), (freeze) and (del). In order to have a deterministic reduction strategy, we assume some arbitrary

rule for selecting one among all the occurences of variables in $\mathsf{dom}(\iota, \rho)$[3], that is, for decomposing a value containing free variables in $\mathsf{dom}(\iota, \rho)$ as $v[x]$.
The effect of rule (res-var) is illustrated by the following example.

$$
\begin{array}{l}
[;\,;y:2 \mid ([;\,;x:1 \mid \lambda z.x+y] + [;Z:0;])\uparrow]\uparrow \overset{\text{(res-var)}}{\longrightarrow} \\
[;\,;y:2 \mid ([;\,;x:1 \mid \lambda z.1+y] + [;Z:0;])\uparrow]\uparrow \overset{\text{(res-extract)}}{\longrightarrow} \\
[;\,;y:2 \mid \lambda z.1+y]\uparrow \overset{\text{(res-var)}}{\longrightarrow} \\
[;\,;y:2 \mid \lambda z.1+2]\uparrow \overset{\text{(res-extract)}}{\longrightarrow} \\
\lambda z.1+2
\end{array}
$$

Relation with CMS and CMS$^\ell$ Apart from the selection operator, *CMS* corresponds to the subset of the calculus obtained by only taking basic modules, module operators (sum, reduct and freeze) and corresponding rules (m-sum), (m-reduct) and (m-freeze). Selection can be simulated by using the run and result operator (see Section 4.1). *CMS*$^\ell$corresponds to the subset obtained by taking basic modules and module operators, basic configurations and the run operator in a non higher-order setting (that is, components of modules and configurations can only be core terms), hence there is no result operator. Moreover, no access to virtual variables is supported (formally, there is no rule (virtual)). This leads to a confluent calculus under the hypothesis that the core calculus is confluent as well; in the calculus presented in this paper, instead, since definition of components may change by performing module operators, there are potentially different results depending on the time when module operators is performed. Hence, it is important to fix (and to assume at the core level as well) a deterministic strategy.

4 Expressive Power of the Calculus

In this section we show that *CMS*$^{\ell,v}$ is much more expressive than *CMS* and *CMS*$^\ell$, and illustrate how it could serve as a formal basis for modeling some interesting mechanisms like marshaling and dynamic software update.

4.1 Module Selection

Module selection [13, 23, 21, 7] allows the users to execute module components from the outside. Conventionally, this operation is permitted only for closed modules in order to avoid scope extrusion of variables which would lead to dynamic errors. For instance, in the ML-like module systems, selection is allowed for structures but not for functors. In *CMS* selection takes the usual syntactic form $e.X$, where e is a module expression and X is a component name. The corresponding reduction rule can be applied only when e is a basic mixin module $[\iota; o; \rho]$ where ι is empty (hence, the module is closed), and X is in the domain

[3] For instance, the leftmost innermost occurrence.

of o. If so, the corresponding expression $o(X)$ is extracted out of the module, and all variables in the domain of ρ possibly occurring free in $o(X)$ are replaced, following the usual unfolding semantics for mutually recursive declarations.
CMS selection can be encoded as a derived operation in $CMS^{\ell,v}$ by means of the $\_\downarrow_X$ and $\_\uparrow$ operators.
Consider for instance, the *CMS* expression $[;X:x;x:1].X$, where we select an output component from a basic module. This expression can be encoded in $CMS^{\ell,v}$ as $[;X:x;x:1]\downarrow_X\uparrow$, which in one step reduces to the basic configuration $[;X:x;x:1 \mid x]\uparrow$. In this way, the defining expression of the selected component can be executed within the context offered by other definitions inside the module and extracted only when it does no longer depends on them. In contrast to *CMS*, this semantics definition, besides being more perspicuous (no unfolding is needed), allows selection of open modules. For example, the expression $[y:Y;Z:y,X:x;x:1].X$ is stuck in *CMS*, while here reduces to the expected value 1.

4.2 Static and Dynamic Rebinding of Virtual Components

The *CMS* calculus supports redefinition of virtual components, a feature analogous to method overriding in object-oriented languages. To see this, let us consider a simple example written in a hypothetical module language with virtual components, whose semantics can be easily expressed in terms of $CMS^{\ell,v}$.

```
M1 = module {
 virtual X=1;
 virtual Y=X+1;
}
```

Here `X` and `Y` are the names of the two externally visible components of `M1`. The semantics of `M1` is given by translation into the following basic module:

$$M_1 = [x:X, y:Y; X:1, Y:x+1;]$$

As already explained in the previous section, the two components X and Y cannot be selected in *CMS* as they are, but in order to do that, they first need to be frozen with the $\mathsf{freeze}_\_\_$ operator which permanently binds their values to the corresponding variables x and y (which become local).

$$\mathsf{freeze}_X\mathsf{freeze}_Y[x:X, y:Y; X:1, Y:x+1;] \longrightarrow [;X:1, Y:x+1; x:1, y:x+1]$$

Then, X and Y can be selected obtaining respectively 1 and 2, as expected.
However, before being frozen, virtual components can be redefined by means of the overriding operator, which can be expressed as a combination of the delete and sum operators at the lower level. For instance, the expression

```
M2 = M1 <- module {virtual X=2;}
```

translates into the lower level expression

$$M_2 = M_1 \setminus_X + [x : X; X : 2;]$$

which reduces to $[x : X, x' : X, y : Y; X : 2, Y : x + 1;]$.
After freezing, if we select Y, then we get 3 rather than 2; hence, the modification of the virtual component X has affected Y as well, whose definition depends on X. In other words, the variable x associated with X has been *rebound*. However, in *CMS* such a rebinding is always *static* rather than *dynamic*, in the sense that it can never happen that a variable of a module is rebound during the execution of a component of the same module.
In fact, in *CMS* the module operators model static configuration of software fragments (as the conventional static linking), whereas selection corresponds to execution, and there is no way to interleave configuration and execution phases for a given module. In *CMS*$^\ell$ linking can take place at execution time, but the program cannot use virtual components. Hence a needed component must be linked in order to be available, and then there is no way to change its definition.
In contrast, *CMS*$^{\ell,v}$ supports *dynamic* rebinding of virtual components. This is possible because execution and configuration phases can be interleaved, and the program can use virtual components.
For instance, consider the following expression (in the higher level language):

```
result(module { E=X+Y+X; virtual X=1; } with main E <-
       module { virtual X=2; virtual Y=X+1; })
```

where the left hand side of the overriding operator `<-` is a configuration whose program is the non virtual (that is, frozen) component `E`, the right hand side lazily overrides the configuration, and `result` is the higher level syntax for the operator $\_\uparrow$. By considering the corresponding translation at the lower level, the reader may verify that the first occurrence of `X` in the definition of `E` reduces to 1, whereas the second to 2, and that the overall expression reduces to 6.

$$[x : X, y : Y; X : 1, E : x + y + x;] \downarrow_E \setminus_X + [x : X, y : Y; X : 2, Y : x + 1;] \uparrow$$

4.3 Dynamic Rebinding for Marshaling and Update

Since *CMS*$^{\ell,v}$ supports dynamic rebinding, it provides a natural formal basis for modeling marshaling and update.
Consider again an example in our hypothetical higher level language:

```
M3 = module {
  virtual X=1;
          Y=2;
          Z=3;
}
with main marshal X+Y+X+Z rebind Y;
```

In the definition of the main expression of `M3`, the expression to be marshaled depends on three different components, already defined in the scope of the main; however, when marshaling an expression e, the user may specify a list of components which have to be rebound when e will be eventually unmarshaled. In this specific case, for correctly unmarshaling the value returned by the execution of `M3`, a new definition for `Y` must be provided, whereas for `X`, `Z` this is left to choice, as in the following example:

```
M2=unmarshal result(M3) bind Y:4,X:5,Z:6;
```

We can now show how the marshal and unmarshal expressions above could be translated into the lower level calculus $CMS^{\ell,v}$. For marshaling we have:

$$e_3 = \mathsf{marshal}([x : X, y : Y; X : 1, Z : z; z : 3 \mid x + y + x + z])$$

The translation is based on the following basic idea: the expression e to be marshaled is packaged with a basic module into a configuration $[\iota; o; \rho \mid e']$, where e' is a suitable translation of e, and $[\iota; o; \rho]$ is obtained from the current context by making deferred all components which have to be rebound. Then, the $\mathsf{marshal}$ constructor can be applied to the resulting configuration.[4]
In the running example, the module corresponding to the current context of the `main` expression is

$$[x : X; X : 1, Y : y, Z : z; y : 2, z : 3]$$

However, since `Y` must be rebound, its definition is removed and its variable becomes deferred.
For unmarshaling, the corresponding lower level expression is:

$$e_2 = (\mathsf{unmarshal}(e_3)\backslash_X \backslash_Z + [; Y : 4, X : 5, Z : 6;])\uparrow$$

where $[; Y : 4, X : 5, Z : 6;]$ is obtained from the binding specified by the unmarshal operator. Since e_3 is closed, $\mathsf{unmarshal}(e_3)$ reduces to
$[x : X, y : Y; X : 1, Z : z; z : 3 \mid x + y + x + z]$. Therefore e_2 reduces to

$$([x : X, y : Y; X : 1, Z : z; z : 3 \mid x + y + x + z]\backslash_X \backslash_Z + [; Y : 4, X : 5, Z : 6;])\uparrow$$

Now, in the expression $x + y + x + z$, the first occurrence of x is bound to 1; then, since y is needed, the delete and sum operators are performed, hence y is bound to 4, and the value of X is overriden, hence the second occurrence of x is bound to 5. Finally, z is bound to 3 (the overriding of Z has no effect).
The example illustrates that the representation of marshaled values as $CMS^{\ell,v}$ configurations allows to code in a natural way different requirements for unmarshaling. If there is an explicit `rebind` directive in marshaling, as for `Y`, then `Y` must be provided since it is undefined (deferred) in the marshaled expression. If

[4] The constructor $\mathsf{marshal}$ and the corresponding destructor $\mathsf{unmarshal}$ must be introduced in the lower level calculus for distinguishing between marshaled and ordinary values.

there is no `rebind` directive, as for `X` and `Z`, then the latest available versions of `X` and `Z` can be provided in order to update the marshaled code in case it contains obsolete versions. However, while the update of `X` (which is virtual) might be reflected into a rebinding of some occurrence of `X` inside the unmarshaled expression, the update of `Z` (which is frozen) has no effects on the evaluation of the inner expression; this is an import feature which provides a protection mechanism against unwanted software update. Finally, note that the update of `X` is lazy (only the second occurrence of x is updated).
We have shown above just some simple examples; the definition of a worked out higher-level language based on $\mathit{CMS}^{\ell,v}$ with marshaling and unmarshaling operators, including more convenient and practical mechanisms for obtaining the configuration to be packaged with the marshaled expression, as the **mark** operator in [10], remains an important subject of further work. However, we believe the examples above are enough to give the flavour of how marshaling mechanisms (where the expression to be marshaled needs to be packaged together with some of the currently available bindings, and needs to be abstracted w.r.t. the components that have to be rebound) could be expressed in a natural way by the notions of basic module (abstractions plus bindings) and configuration (expression packaged with a basic module) provided by $\mathit{CMS}^{\ell,v}$.

5 Type System

In this section we present a type system for $\mathit{CMS}^{\ell,v}$ which prevents reduction from getting stuck.
Types have the following form:

$$\begin{array}{ll} \tau \in \mathsf{Type} & ::= c\tau \mid [\pi^\iota; \pi^o; \tau^\bullet] \\ \tau^\bullet \in \mathsf{Type}^\bullet & ::= \tau \mid \bullet \end{array}$$

Core types are ranged over by $c\tau$. Module types are as in *CMS*, that is, pairs $[\pi^\iota; \pi^o; \bullet]$ where $\pi^\iota, \pi^o : \mathsf{Name} \overset{fin}{\to} \mathsf{Type}$ are the *input* and *output signature*, respectively. Configuration types have the form $[\pi^\iota; \pi^o; \tau]$: the first two components have the same meaning as for module types, whereas τ is the type of the program running in the configuration.
Fig.4 gives the typing rules for deriving judgments of the form $\Gamma \vdash e : \tau$, meaning "e is a well-formed expression of type τ in the environment Γ", where $\Gamma : \mathsf{Var} \overset{fin}{\to} \mathsf{Type}$.
The definition of the type system is parametric in the typing rules for the core level.
In rule (m-basic) and (basic), $\_[\_]$ denotes environment updating. In the side-condition of these rules, we check that virtual names have the same types in the input and the output signatures (recall that the notation $f|g$ means that f and g agree on the common domain).
The (sum) typing rules allow sharing of input components having the same name and type, while preventing output components from being shared (recall

… (rules for core operators)

$$\text{(m-basic)}\ \frac{\begin{array}{c}\{\Gamma[\Gamma^\iota,\Gamma^\rho]\vdash o(X):\pi^o(X)\mid X\in\mathsf{dom}(o)\}\\ \{\Gamma[\Gamma^\iota,\Gamma^\rho]\vdash \rho(x):\Gamma^\rho(x)\mid x\in\mathsf{dom}(\rho)\}\end{array}}{\Gamma\vdash[\iota;o;\rho]:[\pi^\iota;\,\pi^o;\,\bullet]}\ \begin{array}{l}\mathsf{dom}(\pi^\iota)=\mathsf{img}(\iota)\\ \mathsf{dom}(\pi^o)=\mathsf{dom}(o)\\ \Gamma^\iota=\pi^\iota\circ\iota\\ \mathsf{dom}(\Gamma^\rho)=\mathsf{dom}(\rho)\\ \pi^\iota|\pi^o\end{array}$$

$$\text{(basic)}\ \frac{\begin{array}{c}\{\Gamma[\Gamma^\iota,\Gamma^\rho]\vdash o(X):\pi^o(X)\mid X\in\mathsf{dom}(o)\}\\ \{\Gamma[\Gamma^\iota,\Gamma^\rho]\vdash \rho(x):\Gamma^\rho(x)\mid x\in\mathsf{dom}(\rho)\}\quad \Gamma[\Gamma^\iota,\Gamma^\rho]\vdash e:\tau\end{array}}{\Gamma\vdash[\iota;o;\rho\mid e]:[\pi^\iota;\,\pi^o;\,\tau]}\ \begin{array}{l}\mathsf{dom}(\pi^\iota)=\mathsf{img}(\iota)\\ \mathsf{dom}(\pi^o)=\mathsf{dom}(o)\\ \Gamma^\iota=\pi^\iota\circ\iota\\ \mathsf{dom}(\Gamma^\rho)=\mathsf{dom}(\rho)\\ \pi^\iota|\pi^o \text{ and } \vdash\tau\diamond\end{array}$$

$$\text{(sum)}\ \frac{\begin{array}{c}\Gamma\vdash e_1:[\pi^\iota{}_1;\,\pi^o{}_1;\,\tau^\bullet]\\ \Gamma\vdash e_2:[\pi^\iota{}_2;\,\pi^o{}_2;\,\bullet]\end{array}}{\Gamma\vdash e_1+e_2:[\pi^\iota{}_1\cup\pi^\iota{}_2;\,\pi^o{}_1,\pi^o{}_2;\,\tau^\bullet]}\ \pi^\iota{}_1\cup\pi^\iota{}_2|\pi^o{}_1,\pi^o{}_2$$

$$\text{(del)}\ \frac{\Gamma\vdash e:[\pi^\iota;\,\pi^o;\,\tau^\bullet]}{\Gamma\vdash e\backslash_X:[\pi^\iota;\,\pi^o\backslash_X;\,\tau^\bullet]}\ X\in\mathsf{dom}(\pi^o)$$

$$\text{(freeze)}\ \frac{\Gamma\vdash e:[\pi^\iota;\,\pi^o;\,\tau^\bullet]}{\Gamma\vdash \mathsf{freeze}_X e:[\pi^\iota{}_2\backslash_X;\,\pi^o;\,\tau^\bullet]}\ X\in\mathsf{dom}(\pi^\iota)\Rightarrow X\in\mathsf{dom}(\pi^o)$$

$$\text{(run)}\ \frac{\Gamma\vdash e:[\pi^\iota;\,\pi^o;\,\bullet]}{\Gamma\vdash e{\downarrow_X}:[\pi^\iota;\,\pi^o;\,\pi^o(X)]}\ \vdash\pi^o(X)\diamond$$

$$\text{(res)}\ \frac{\Gamma\vdash e:[\pi^\iota;\,\pi^o;\,\tau]}{\Gamma\vdash e{\uparrow}:\tau}\ \vdash[\pi^\iota;\,\pi^o;\,\tau]\ \diamond$$

Fig. 4. Typing rules

that $f_1 \cup f_2$ denotes the union of two compatible partial functions, while f_1, f_2 denotes the union of two maps with disjoint domain). Moreover, we check that names that will become virtual performing the sum will have the same types in both the (resulting) input and the output signatures.
In rule (basic) and (res) the judgment $\vdash \tau \diamond$ means "τ is a closed type". A closed type is either a core or module type, or a configuration type with no deferred components, as formally defined in Fig.5.

$$\frac{}{\vdash c\tau \diamond} \qquad \frac{}{\vdash [\pi^\iota;\, \pi^o;\, \bullet]\ \diamond} \qquad \frac{}{\vdash [\pi^\iota;\, \pi^o;\, \tau]\ \diamond}\ \pi^\iota \subseteq \pi^o$$

Fig. 5. Closed types

Intuitively, (ground) terms of closed types are those which can be safely used in isolation, since they do not depend on any missing variable or component. Formally, we state the progress property only on these terms. The reason for requiring that the program in a basic configuration and the argument of a result operator are of closed type is that in both cases the term is inserted in a context where no more reconfiguration operators are applied, hence, in case it is a configuration term whose program needs a deferred variable, this will never be provided. For instance, the term $[y : Y; ; \mid y]$ is a well-typed term of (non-closed) type, which can be for instance inserted in the context $\Box + [; Y : 0;]$ giving a safe term which reduces to the value $[y : Y; Y : 0; \mid 0]$. However, the terms of the form $[\iota; o; \rho \mid [y : Y; ; \mid y]]$ and the term $[y : Y; ; \mid y] \uparrow$ are ill-formed since they give a stuck computation in whichever context they are inserted, since there is no way to provide component Y to the program.

6 Results

In this section we illustrate the properties of the type system of $\mathit{CMS}^{\ell,v}$. For space limitations, all proof have been omitted, together with the results on the determinacy of the reduction relation; however, they are available in an extended version of this paper.[5]
In general, all the results we state hold under the assumption that (roughly speaking) they are verified at the core level as well. This assumption is formally detailed for each case.
The type system guarantees that the reduction relation does not get stuck on ground terms of closed type (progress property) and preserves types (subject reduction property).
In order to prove these results, we need the following lemmas, which can be proved by induction on the typing rules under the assumption that, for each

[5] `ftp://ftp.disi.unige.it/pub/person/AnconaD/MMDRlong.pdf`

core typing rule, if the property holds for the premises, then it holds for the consequence as well.

Lemma 1 (Weakening). *If $\Gamma \vdash e : \tau$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash e : \tau$.*

Lemma 2 (Strengthening). *If $\Gamma \vdash e : \tau$, $\Gamma' \subseteq \Gamma$, and $\mathsf{FV}(e) \subseteq \mathsf{dom}(\Gamma')$, then $\Gamma' \vdash e : \tau$.*

Lemma 3 (Substitution). *If $\Gamma \vdash \mathcal{E}[x] : \tau$, $\Gamma(x) = \tau_x$ and $\Gamma \vdash e : \tau_x$, then $\Gamma \vdash \mathcal{E}\{e\} : \tau$.*

Lemma 4 (Canonical Forms). *Given $v \in \mathsf{Val}$,*

- *if $\Gamma \vdash v : c\tau$, then v is a core value;*
- *if $\Gamma \vdash v : [\pi^\iota; \pi^o; \bullet]$, then v has the form $[\iota; o; \rho]$;*
- *if $\Gamma \vdash v : [\pi^\iota; \pi^o; \tau]$, then v has the form $\mathcal{R}[\iota; o; \rho \mid v']$, and $\Gamma[\Gamma^\iota][\Gamma^\rho] \vdash v' : \tau$, with $\mathsf{dom}(\Gamma^\iota) = \mathsf{dom}(\iota)$ and $\mathsf{dom}(\Gamma^\rho) = \mathsf{dom}(\rho)$.*

In the standard formulation, soundness of a type system is shown by separately proving subject reduction and progress property. Subject reduction (preservation of type under reduction) holds for all well-typed terms, whereas progress only holds for terms which can be seen as "executable", that is, can be safely reduced in isolation. Usually, executable terms correspond to ground terms, that is, terms without free variables. Terms with free variables represent open code fragments, which cannot be safely reduced, but are still well-typed since they can be safely used as subterms of an executable program.
In $CMS^{\ell,v}$, the progress property holds on terms that are not only ground, but also of a closed type, that is, a type with no deferred components. However, terms of non-closed types are still well typed, since they can be inserted inside contexts providing all needed components.

Theorem 1 (Subject Reduction). *If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$ under the assumption that, for each core reduction rule, if the property holds for the premises, then it holds for the consequence as well.*

The progress property follows as a corollary of a *generalized progress* property, which states that a well-typed term can get stuck for two reasons: either it contains some free variable (in which case, intuitively, execution could proceed by replacing this variable) or it is a basic configuration whose program needs a deferred component which is not available (in which case, intuitively, execution could proceed by providing this component.)

Theorem 2. *If $\Gamma \vdash e : \tau$, then one of the following cases holds*

- $e \in \mathsf{Val}$
- $e \longrightarrow e'$, *for some* $e' \in \mathsf{Exp}$,
- $e = \mathcal{E}[x]$, $x \notin \mathsf{HB}(\mathcal{E})$, $x \in \mathsf{dom}(\Gamma)$,
- $e = [\iota; o; \rho \mid \mathcal{E}[x]]$, *with* $x \notin \mathsf{HB}(\mathcal{E}), x \in \mathsf{dom}(\iota)$ *and* $\iota(x) \notin \mathsf{dom}(o)$
  under the assumption that, for each core typing rule, if the property holds for the premises, then it holds for the consequence as well.

Corollary 1 (Progress). *If $\emptyset \vdash e : \tau$ and $\vdash \tau \diamond$, then either $e \in \mathsf{Val}$ or $e \longrightarrow e'$, for some $e' \in \mathsf{Exp}$.*

7 Conclusion

We have presented a module calculus $CMS^{\ell,v}$ which allows to express in a natural way rebinding through the notion of *virtual component*, and to make this rebinding dynamic by allowing standard program execution to be interleaved with reconfiguration steps. We have illustrated the expressive power of the calculus and provided a sound type system.

This work is part a stream of research [4, 6, 5, 16] whose aim is the development of foundational calculi providing an abstract framework for dynamic software reconfiguration. In particular, the possibility of extending module calculi with selection on open modules, interleaving of component evaluation with reconfiguration steps and a lazy strategy has been firstly explored in [6]. As already explained, the calculus presented in this paper contains two key novelties.

First, $CMS^{\ell,v}$ allows the executing program to use *virtual* variables; this provides a natural mechanism for rebinding, which greatly enhances the expressive power. Indeed, execution can refer to components whose definition may change by performing module operators, leading potentially to different results depending on the time when module operators is performed, that is, before or after accessing a virtual variable. This is avoided here by taking a deterministic strategy which performs substitution of local/virtual variables and resolution (by reconfiguration steps) of deferred variables only on demand.

Then, higher-order configurations, together with the run and result operators, allow to express interaction of execution at different levels (e.g., modules with module or configuration components, starting a local configuration level inside program execution, a scoping mechanism for nested variable resolution and triggering of reconfiguration steps).

In [5] we have investigated how to increase flexibility in a different direction, that is, by allowing a limited form of swapping between module operators. Finally, Fagorzi's thesis [16] provides a comprehensive presentation of most part of this work, and, moreover, the definition of a pure[6] reconfiguration calculus called R, in two versions which either allow or not to use virtual variables. This calculus is confluent in the non-virtual version, and a comparative discussion on different possible type systems is also given.

On the theoretical side, the ideas presented in this paper look similar to those at the basis of literature on laziness in functional calculi (see, e.g., [8]) and dynamic binding. In particular, some recent work on dynamic rebinding [10] presents a call-by-value λ-calculus which delays instantiation of identifiers, in such a way that computations can use the most recent versions of rebound definitions. It is well-known that record-based calculi can provide an alternative computational paradigm where λ-calculus can be encoded [1, 7]. In our work, we are firstly exploring laziness (obtained by delaying record composition after selection) in this alternative paradigm. The advantages offered by the record-based paradigm are a natural syntactic representation of a scope (a record, or basic module in the terminology of this paper) and a built-in mechanism for rebinding (by deleting

[6] That is, with no fixed strategy.

and then adding record component) without any need of introducing imperative features at the core level.
Hence, a very interesting subject of further work is a formal comparison with laziness obtained by delaying application in functional calculi. A preliminary attempt in this direction is in [17], where we outline a call-by-need strategy for R(in the non-virtual version) which smoothly generalizes the approach in [8] where an expression is evaluated the first time it is needed and only once.
On a more applicative side, though the area of unanticipated software evolution continues attracting large interest, with its foundations studied in, e.g., [22], there is a little amount of work going toward the development of abstract models for dynamic linking and updating. Apart from the wide literature concerning concrete dynamic linking mechanisms in existing programming environments [14, 15], we mention [9], 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, [2], where dynamic linking is studied as the programming language counterpart of the axiom of choice, and the module system defined in [20], where static linking, dynamic linking and cross-computation communication are all defined in a uniform framework.
Further work includes, as already mentioned, a deeper investigation of the relation with lazy lambda-calculi, and the further development of the techniques for encoding dynamic rebinding, marshaling and update outlined in Sect.4. The expressive power of lazy module calculi should also be analyzed by showing which kind of real-world reconfiguration mechanisms can be modeled and which kind require a richer model. Finally, an important issue is the integration with mobility aspects, that is, the design of calculi for reconfiguration where, roughly speaking, code to be used for reconfiguring the running program can migrate from a different process.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1996.
2. Martin Abadi, Georges Gonthier, and Benjamin Werner. Choice in dynamic linking. In *FOSSACS'04 - Foundations of Software Science and Computation Structures 2004*, Lecture Notes in Computer Science, pages 12–26. Springer, 2004.
3. D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, and E. Zucca. A type preserving translation of Fickle into Java. *Electonical Notes in Theoretical Computer Science*, 62, 2002.
4. 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.
5. D. Ancona, S. Fagorzi, and E. Zucca. A calculus for dynamic reconfiguration with low priority linking. *Electonical Notes in Theoretical Computer Science*, 2004. In WOOD'04: Workshop on Object-Oriented Developments. To appear.
6. D. Ancona, S. Fagorzi, and E. Zucca. A calculus with lazy module operators. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *TCS 2004 (IFIP*

Int. Conf. on Theoretical Computer Science), pages 423–436. Kluwer Academic Publishers, 2004.

7. D. Ancona and E. Zucca. A calculus of module systems. *Journ. of Functional Programming*, 12(2):91–132, 2002.
8. Z. M. Ariola and M.Felleisen. The call-by-need lambda calculus. *Journ. of Functional Programming*, 7(3):265–301, 1997.
9. G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating (Extended Abstract). In *USE'03 - the Second International Workshop on Unanticipated Software Evolution*, 2003.
10. G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In C. Runciman and O. Shivers, editors, *Intl. Conf. on Functional Programming 2003*, pages 99–110. ACM Press, 2004.
11. G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
12. L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, 1997.
13. L. Cardelli and X. Leroy. Abstract types and the dot notation. Technical Report 56, DEC SRC, 1990.
14. 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.
15. 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.
16. S. Fagorzi. *Module Calculi for Dynamic Reconfiguration*. PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2005.
17. S. Fagorzi and E. Zucca. A calculus for reconfiguration. In *DCM 2005 - International Workshop on Developments in Computational Models*, July 2005. To appear.
18. Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219, Ebberup, Denmark, August 1986.
19. 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.
20. Y. D. Liu and S. F. Smith. Modules with interfaces for dynamic linking and communication. In M. Odersky, editor, *ECOOP'04 - Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, pages 414–439. Springer, 2004.
21. E. Machkasova and F.A. Turbak. A calculus for link-time compilation. In *ESOP 2000 - European Symposium on Programming 2000*, number 1782 in Lecture Notes in Computer Science, pages 260–274. Springer, 2000.
22. Tom Mens and Guenther Kniesel. Workshop on foundations of unanticipated software evolution. ETAPS 2004, http://joint.org/fuse2004/, 2004.
23. 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.