A Calculus of Module Systems

DAVIDE ANCONA and ELENA ZUCCA*

Dipartimento di Informatica e Scienze dell'Informazione
Via Dodecaneso, 35,16146 Genova (Italy)
email: davide,zucca@disi.unige.it
fax: +39 010-3536699

Abstract

We present *CMS*, a simple and powerful calculus of modules supporting mutual recursion and higher order features, which can be instantiated over an arbitrary core calculus satisfying standard assumptions.

The calculus allows expression of a large variety of existing mechanisms for combining software components, including parameterized modules similar to ML functors, extension with overriding as in object-oriented programming, mixin modules and extra-linguistic mechanisms like those provided by a linker. Hence *CMS* can be used as a paradigmatic calculus for modular languages, in the same spirit the lambda calculus is used for functional programming.

We first present an untyped version of the calculus and then a type system; we prove confluence, progress, and subject reduction properties. Then, we define a derived calculus of mixin modules directly in terms of *CMS* and show how to encode other primitive calculi into *CMS* (the lambda calculus and the Abadi-Cardelli object calculus). Finally, we consider the problem of introducing a subtype relation for module types.

Introduction

Considerable effort has been recently invested in studying theoretical foundations and developing new forms of module systems; let us mention the wide literature about foundations and improvements of the Standard ML (Milner *et al.*, 1990) modules system (see e.g. (Leroy, 1994; Harper & Lillibridge, 1994)), the notions of *mixins* (see e.g. (Bracha, 1992; Duggan & Sourelis, 1996; Flatt *et al.*, 1998) and our previous work (Ancona & Zucca, 1998a; Ancona & Zucca, 1998b)) and *units* (Flatt & Felleisen, 1998), the type-theoretical analysis of recursion between modules proposed in (Crary *et al.*, 1999).

Two principles which seem to emerge as common ideas of all these approaches are the following.

First, a module system should have two linguistic levels, a *module language* providing operators for combining software components, constructed on top of a *core language*[1] for defining module components. The module language should have its

* Partially supported by Murst - Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software and APPlied SEMantics - Esprit Working Group 26142.

[1] Following the terminology introduced with Standard ML.

own typing rules and be as independent as possible from the core language; even more, it could be in principle instantiated over different core languages (see (Leroy, 2000) for an effective demonstration).

Second, modules should actually correspond to compilation units, and typing rules of the module language should formalize the *inter-check phase* described in (Cardelli, 1997). Indeed, operators of the module language could also correspond, in practice, to an extra-linguistic tool like a linker.

In this paper, we define a primitive module calculus based on these two principles and suitable for encoding various existing mechanisms for composing modules, in the same way as λ-calculus provides a theoretical basis for functional languages; in particular it supports mutually recursive modules and higher-level features (modules with module components), and it is parametric in the underlying core language.

A basic module of this calculus is written, using some syntactic sugar and considering here for simplicity the untyped version, as follows:

`import` X_1 `as` x_1`, ..,` X_m `as` x_m
`export` Y_1 `=` E_1`, ..,` Y_n `=` E_n
`local` z_1 `=` E'_1`, ..,` z_p `=` E'_p

A basic module introduces *component names* and *variables*. The former are the names of the components the module either *imports* from (*input* components X_1, .., X_m) or *exports* to (*output* components Y_1, .., Y_n) the outside. The latter are the variables used in definitions inside the module (that is, the expressions E_1, .., E_n, E'_1, .., E'_p, which can be expressions of the core language or in turn module expressions if the module has module components). These variables can be either *deferred* (x_1, .., x_n), that is, associated with some input component, or locally defined (z_1, .., z_p). This distinction between component names and variables is essential for keeping the module level independent from the core level, as will be explained in more detail later.

Now, as example of a typical operator which can be easily encoded in our calculus, consider a *link* operator used for merging two or more modules. This operator may be regarded as either an operation provided by a module language in order to define structured module expressions or an extra-linguistic mechanism to combine object files provided by a tool for modular software development. Regardless of the view we take, we can informally define this operator as follows. For any pair of modules M_1 and M_2, $\mathit{link}(M_1, M_2)$ is well-defined if the sets of the output components of M_1 and M_2 are disjoint. In this case, $\mathit{link}(M_1, M_2)$ corresponds to a module where some input component of one module has been bound to the definition of the corresponding output component of the other module, and conversely. In particular, if each input component of both modules has been bound we get a module with no input components, called a *concrete* module.

For instance, let the modules `BOOL` and `INT` define the evaluation of some boolean and integer expressions in a mutually recursive way:

```
module BOOL is
  import IntEv as ext_ev
  export BoolEv = ev
```

```
  local
   fun ev EQ(ie1,ie2) = ext_ev(ie1)==ext_ev(ie2)
       |  ...
   ...
end BOOL;

module INT is
  import BoolEv as ext_ev
  export IntEv = ev
  local
   fun ev IF(be,ie1,ie2) =  if ext_ev(be) then ev(ie1) else ev(ie2)
       |  ...
   ...
end INT;
```

then *link*(BOOL, INT) intuitively corresponds to the module

```
module BOOL_INT is
  export IntEv = iev
  export BoolEv = bev
  local
   fun bev EQ(ie1,ie2) = iev(ie1)==iev(ie2)
       |  ...
   fun iev IF(be,ie1,ie2) =  ifbev(be) then iev(ie1) else iev(ie2)
       |  ...
   ...
end BOOL_INT;
```

The separation between component names and variables allows one to use internally the same name `ev` for the evaluation function in the two modules; in the compound module, indeed, `ev` of `BOOL` and `ev` of `INT` are α-renamed to `bev` and `iev`, respectively.

Note also that the *link* operator informally described above can be decomposed in two steps: first, just putting together the declarations of the two arguments in one module, yielding

```
module
  import IntEv as ext_iev
  import BoolEv as ext_bev
  export IntEv = iev
  export BoolEv = bev
  local
   fun bev EQ(ie1,ie2) = ext_iev(ie1)==ext_iev(ie2)
       |  ...
   fun iev IF(be,ie1,ie2) =  if ext_bev(be) then iev(ie1) else iev(ie2)
       |  ...
   ...
end;
```

then, binding import components with export components of the same name (in this case), yielding BOOL_INT. Formally, this corresponds to the fact that *link* is a derived operator which can be expressed by the *sum* and *freeze* basic operators of *CMS*.

In the following, we define the calculus *CMS* (Calculus of Module Systems) where module expressions are either basic modules which are, apart from syntactic sugar, those described above, or constructed by three operators (*sum*, *reduct* and *freeze*); moreover, a selection operator allows one to extract a module component (Sect.1.1). In Sect.1.2 we define a reduction semantics for *CMS* and in Sect.2 we prove that *CMS* satisfies the Church-Rosser property (abbreviated CR in the sequel). In Sect.3 we define a typed version of *CMS* and prove that it enjoys the progress and subject reduction properties. In Sect.4 we illustrate how a derived calculus *DCMS* supporting a link operator, like the one informally introduced above, and *virtual* components can be defined on top of *CMS*. This derived calculus supports, in a general framework of module composition, redefinition of components which is typical of object-oriented programming. To analyse the expressivity of the calculus, in Sect.5 we encode the λ-calculus and the Abadi-Cardelli object calculus into CMS. These encodings also shed light on the relationship between *CMS* modules and both ML-like functors and the objects of object-oriented programming. In Sect.6 we propose an extension of typed *CMS* with subtyping and discuss related problems. Finally, related work is considered in Sect.7, while in the Conclusion we summarize the contribution of the paper and outline further work.

This paper is an extended and improved version of (Ancona & Zucca, 1999). The main novel contributions are the following. On the technical side, the ability of representing core terms containing module expressions independently from the core language and without any knowledge of its syntactic rules is now obtained by means of *explicit substitutions* (Curien *et al.*, 1996). This allows a clean integration of the core and module levels; in particular, Church Rosser, progress and subject reduction properties for *CMS* can be stated and even proved in a nice modular way, assuming the same properties at the core level. On the language side, we present the derived calculus *DCMS*, which directly supports virtual components in Sect.4 and an extension with subtyping in Sect.6.

1 Untyped *CMS*

1.1 Syntax

The abstract syntax of the untyped calculus is given in Fig.1.

We first present the module-level syntax, then discuss the abstraction of the core language.

Lower case meta-variable x ranges over an infinite numerable set *Var* of *variables*, whereas upper case meta-variables X and Y range over an infinite numerable set *Name* of *component names*. This distinction at the level of the calculus reflects, from a practical perspective, the separation that a linker makes between *internal names* (what we call variables) and *external names* (what we call component names).

The meta-variable E ranges over the set of all module expressions (or terms) which is denoted by $\mathcal{E}_E$.

A *basic module* corresponds to building a module by collecting a set of components. A basic module is made up of an assignment of *input names* to *deferred*

$$
\begin{array}{lll}
E ::= x & \mid \text{(variable)} \\
\quad [\iota;\ o;\ \rho] & \mid \text{(basic module)} \\
\quad E_1 + E_2 & \mid \text{(sum)} \\
\quad {}_{\sigma^\iota|}E_{|\sigma^o} & \mid \text{(reduct)} \\
\quad \mathbf{freeze}_{\sigma^f}(E) & \mid \text{(freeze)} \\
\quad C[\rho] & \mid \text{(core expression)} \\
\quad E.X & \text{(selection)} \\
\iota ::= x_i \overset{i \in I}{\mapsto} X_i & (\iota\text{-assignment}) \\
o ::= X_i \overset{i \in I}{\mapsto} E_i & (o\text{-assignment}) \\
\rho ::= x_i \overset{i \in I}{\mapsto} E_i & (\rho\text{-assignment}) \\
\sigma ::= X_i \overset{i \in I}{\mapsto} Y_i, Y_j{}^{j \in J} & \text{(renaming)}
\end{array}
$$

Fig. 1. Abstract syntax of the untyped calculus.

variables (also called ι-assignment), of *expressions* to *output names* (also called o-assignment) and of *expressions* to *local variables* (also called ρ-assignment or substitution); all these assignments have a scope that is delimited by square brackets.

The notation $x_i \overset{i \in I}{\mapsto} X_i$ (I possibly empty) is used for representing the unique surjective and finite map ι s.t. $dom(\iota) = \{x_i \mid i \in I\}$, $cod(\iota) = \{X_i \mid i \in I\}$ and $\iota(x_i) = X_i$ for all $i \in I$. The expression is well-formed only if for any i_1 and i_2 in I, if $i_1 \neq i_2$ then $x_{i_1} \neq x_{i_2}$.

The expression ι_1, ι_2 is well-formed only if $dom(\iota_1) \cap dom(\iota_2) = \emptyset$ and denotes the assignment ι s.t. $dom(\iota) = dom(\iota_1) \cup dom(\iota_2)$, $cod(\iota) = cod(\iota_1) \cup cod(\iota_2)$, $\iota(x) = \iota_1(x)$ for all $x \in dom(\iota_1)$ and $\iota(x) = \iota_2(x)$ for all $x \in dom(\iota_2)$. For application and composition of maps we use the standard notation (see the comments on the reduct operator in Sect.1.2, for further details on map composition).

For sake of simplicity, we identify all expressions representing the same map.

Similar notations and assumptions are used for the other kinds of assignments and for renamings.

Finally, we assume that a basic module expression is well-formed only if the set of deferred and local variables are disjoint ($dom(\iota) \cap dom(\rho)$). The sets of input and output components, however, can have a non empty intersection.

As an example of basic module, the expression

$$[\mathit{ext\_ev} \mapsto \mathit{IntEv};\ \mathit{BoolEv} \mapsto \mathit{ev};\ \mathit{ev} \mapsto ...]$$

corresponds to the module `BOOL` defined in the Introduction.

Before describing the other module operators, we should justify the separation of variables and component names. There are several technical and methodological motivations for this separation.

Technically speaking, variables can be α-converted, in the sense that we can rename (in an appropriate way) the variables of an expression E without changing the observable semantics of E. The same cannot be done for component names (see Sect.1.2). Furthermore, the distinction between component names and variables is crucial for avoiding typing problems related to module extension and hiding (see (Riecke & Stone, 1999) and Sect.3). Finally, if we want the module calculus to be independent from the core level, then component names have to be necessarily independent from the variables of the core language.

Methodologically speaking, this separation is a way of abstracting from the particular programming language a module comes from, even allowing composition of heterogeneous software components; variables correspond to the particular dialect spoken inside each module, whereas names represent a sort of lingua franca which allows modules to talk to each other.

Analogous distinctions are those between program variables and labels that connect fragments in (Harper & Lillibridge, 1994), variables and field/method names in the Abadi-Cardelli object calculus (Abadi & Cardelli, 1996) and names and identifiers in (Leroy, 1994); also in the object calculus defined in (Riecke & Stone, 1999) objects have dictionaries mapping external to internal names and in MzScheme's *units* (Flatt & Felleisen, 1998) imported and exported variables have separate internal (binding) and external (linking) names, and the internal names within a unit can be α-renamed.

Returning to the module operators, modules can be merged together by means of the sum operator.

The reduct operator is an expressive construct for renaming the component names; input and output components are separately renamed via two renamings (see below) σ^ι and σ^o, respectively, which are two finite maps over *Name*.

The freeze operator allows the binding between input and output names; this binding is specified by the renaming σ^f.

Finally, it is possible to access an output component from the outside via the selection operator.

The meta-variable σ ranges over the set of renamings (finite maps over *Name*). The notation $X_i \overset{i\in I}{\mapsto} Y_i, Y_j{}^{j\in J}$ (I or J possibly empty) is used for representing the unique map σ s.t. $dom(\sigma) = \{X_i \mid i \in I\}$, $cod(\sigma) = \{Y_i \mid i \in I \cup J\}$ and $\sigma(X_i) = Y_i$, for all $i \in I$.

A renaming $X_i \overset{i\in I}{\mapsto} Y_i, Y_j{}^{j\in J}$ is well-formed only if $\{Y_i \mid i \in I\}$ and $\{Y_j \mid j \in J\}$ are disjoint sets and for any i_1 and i_2 in I, if $i_1 \neq i_2$ then $X_{i_1} \neq X_{i_2}$ and, similarly, for any j_1 and j_2 in J, if $j_1 \neq j_2$ then $Y_{j_1} \neq Y_{j_2}$.

We introduce the following abbreviations for the reduct: if σ^ι is an inclusion, that is, of the form $X_i \overset{i\in I}{\mapsto} X_i, X_j{}^{j\in J}$, then ${}_{\sigma^\iota|}E_{|\sigma^o}$ is written ${}_{\{X_i | i \in I \cup J\}|}E_{|\sigma^o}$; if in particular $J = \emptyset$, that is, σ^ι is the identity, then we simply write $E_{|\sigma^o}$. Analogously, if σ^o is of the form $X_i \overset{i\in I}{\mapsto} X_i, X_j{}^{j\in J}$, then ${}_{\sigma^\iota|}E_{|\sigma^o}$ is written ${}_{\sigma^\iota|}E_{|\{X_i | i \in I\}}$ and, if σ^o is the identity, then we simply write ${}_{\sigma^\iota|}E$.

The meta-variable C ranges over the set of core expressions (or terms), denoted by $\mathcal{E}_C$. Therefore the syntax of *CMS* is parametric in the set $\mathcal{E}_C$; we assume that $Var \subseteq \mathcal{E}_C$.

The production $E ::= C[\rho]$ intuitively corresponds to the term obtained by applying the substitution ρ to C with no capture of free variables. Note that the substitution is *explicit*, in the sense of (Curien *et al.*, 1996); in other words, $C[x_i \overset{i\in I}{\mapsto} E_i]$ is a core term annotated with a "pending" substitution where arbitrary (that is, even non-core) expressions may appear. The intuition is that, when reducing terms, the explicit substitution $x_i \mapsto E_i$ (for some $i \in I$) can be effectively "applied" to the core term C whenever E_i is a core expression (see rule (sub) in Fig.2 in the sequel). In this way we can represent core terms containing module expressions independently from the core language and without any knowledge of its syntactic rules. Whenever the substitution is empty and no ambiguity arises, we simply write C instead of $C[\,]$.

The ability to represent this kind of terms (that is, core terms containing module subterms) is crucial if we want our module calculus to have a *selection* operator (see Fig.2 and comments below). Indeed, this is the only module operator of *CMS* which can return as result a core term (possibly containing module subterms). If we removed selection from the set of *CMS* operators, then we could replace the production $E ::= C[\rho]$ with $E ::= C$, thus avoiding explicit substitution.

However, we consider selection an essential feature of *CMS*, since no module language is usable in practice without this operator. On the other hand, selection can be disregarded if we are interested only in modeling extra-linguistic mechanisms for combining modules (like linking).

The production $E ::= C[\rho]$ is applicable only under the following implicit assumptions:

(1) $FV_C(C) \subseteq dom(\rho)$
(2) $FV(C[\rho]) \cap dom(\rho) = \emptyset$

where $FV_C\colon \mathcal{E}_C \to \mathcal{P}(Var)$ is a core language dependent function associating with each core expression the set of its free variables; $dom(\rho)$ denotes the domain of ρ (formally defined below); $FV(C[\rho])$ denotes the set of free variables in $C[\rho]$ (formally defined below). These assumptions are needed for proving CR (Sect.2); on the other hand, they do not restrict the expressive power of the language (see the definition of α-congruence below).

C can be considered as is applied to C.

The independence of the calculus from the core language is *effective*, in the sense that reduction and typing rules we provide are constructed on top of those of the core language, so that a type-checker or an interpreter for the module language could be constructed in a modular way enriching one for the core level, as done in (Leroy, 2000). The prototype we have developed for the calculus is actually built following this idea (see the Conclusion).

Note that the approach taken here does not allow the definition of the core language to depend in turn on the module language, that is, having a mutual dependency between the module and the core language. However, our prototype, which has been written in Java, goes further and allows the user to define instantiations of *CMS* where modules are first class values at the core level (at least for the untyped version). We leave to further investigation the possibility of defining a formal framework allowing mutual dependency between the core and the module level (see the Conclusion).

Notations and Definitions: for any module expression E, let $FV(E)$ denotes the set of free variables of E inductively defined by:

$FV(x) = \{x\}$
$FV(C[\rho]) = \bigcup_{E \in cod(\rho)} FV(E)$
$FV([\iota;\ o;\ \rho]) = \bigcup_{E \in cod(o) \cup cod(\rho)} FV(E) \setminus (dom(\iota) \cup dom(\rho))$
$FV(E_1 + E_2) = FV(E_1) \cup FV(E_2)$
$FV({}_{\sigma^\iota|}E_{|\sigma^o}) = FV(E)$
$FV(\mathbf{freeze}_{\sigma^f}(E)) = FV(E)$
$FV(E.X) = FV(E)$

As expected, at the module level the only binding construct is that for basic modules. If $E = [\iota;\ o;\ \rho]$ then we denote by $BV(E)$ the set $dom(\iota) \cup dom(\rho)$ of its binders. Finally, we define $V(E)$ to be the set of all variables in E.

Capture avoiding and parallel meta-level substitution for module terms is inductively defined as follows:

$$
\begin{array}{ll}
x\{\rho\} = \rho(x) & \text{if } x \in dom(\rho) \\
x\{\rho\} = x & \text{if } x \notin dom(\rho) \\
\multicolumn{2}{l}{C[x_i \overset{i \in I}{\mapsto} E_i]\{\rho\} = C[x_i \overset{i \in I}{\mapsto} E_i\{\rho\}]} \\
\multicolumn{2}{l}{[\iota;\ o;\ \rho]\{\rho'\} = [\iota;\ o\{\rho'_{|V}\};\ \rho\{\rho'_{|V}\}],\ V = dom(\rho') \setminus BV([\iota;\ o;\ \rho])} \\
 & \text{if } BV([\iota;\ o;\ \rho]) \cap \bigcup_{E \in cod(\rho')} FV(E) = \emptyset \\
\multicolumn{2}{l}{(E_1 + E_2)\{\rho\} = E_1\{\rho\} + E_2\{\rho\}} \\
\multicolumn{2}{l}{{}_{\sigma^\iota|}E_{|\sigma^o}\{\rho\} = {}_{\sigma^\iota|}E\{\rho\}_{|\sigma^o}} \\
\multicolumn{2}{l}{\mathbf{freeze}_{\sigma^f}(E)\{\rho\} = \mathbf{freeze}_{\sigma^f}(E\{\rho\})} \\
\multicolumn{2}{l}{E.X\{\rho\} = E\{\rho\}.X} \\
\multicolumn{2}{l}{(X_i \overset{i \in I}{\mapsto} E_i)\{\rho\} = X_i \overset{i \in I}{\mapsto} E_i\{\rho\}} \\
\multicolumn{2}{l}{(x_i \overset{i \in I}{\mapsto} E_i)\{\rho\} = x_i \overset{i \in I}{\mapsto} E_i\{\rho\}}
\end{array}
$$

where $\rho'_{|V}$ denotes ρ' restricted to the domain V ($V \subseteq dom(\rho')$). The condition for the case of basic modules ensures that there are no captures of free variables; it is sufficient, but clearly not necessary.

Note that the definition of meta-level substitution for module terms is completely independent of that for core terms (see below).

α-congruence: we define the relation $\cong$ over well-formed module terms as the congruence inductively defined by the following rules:

(core) $$\frac{C \cong_C C'}{C[\rho] \cong C'[\rho]}$$

(sub) $$\frac{}{C[x \mapsto E, \rho] \cong C\{x \mapsto x'\}[x' \mapsto E, \rho]}$$

(ι-basic) $$\frac{}{[x \mapsto X, \iota;\ o;\ \rho] \cong [x' \mapsto X, \iota;\ o\{x \mapsto x'\};\ \rho\{x \mapsto x'\}]}$$

(ρ-basic) $$\frac{}{[\iota;\ o;\ x \mapsto E, \rho] \cong [\iota;\ o\{x \mapsto x'\};\ x' \mapsto E\{x \mapsto x'\}, \rho\{x \mapsto x'\}]}$$

The definition of $\cong$ depends on two "ingredients" that have to be provided by the core calculus: α-congruence $\cong_C$ and capture avoiding substitution over core terms (differently from substitution for module terms, we do not need parallel substitution at the core level).

$$\text{(core)}\quad \frac{C \overset{+}{\rightarrow}_C C'}{C[\rho] \rightarrow C'[\rho]}$$

$$\text{(sub)}\quad \frac{}{C[x \mapsto C'[\rho_1], \rho_2] \rightarrow C\{x \mapsto C'\}[\rho_1, \rho_2]}$$

$$\text{(sum)}\quad \frac{}{[\iota_1;\ o_1;\ \rho_1] + [\iota_2;\ o_2;\ \rho_2] \rightarrow [\iota_1, \iota_2;\ o_1, o_2;\ \rho_1, \rho_2]}\quad \begin{array}{l} BV(E_1) \cap FV(E_2) = \emptyset \\ BV(E_2) \cap FV(E_1) = \emptyset \\ E_i = [\iota_i;\ o_i;\ \rho_i],\ i = 1,2 \end{array}$$

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

$$\text{(freeze)}\quad \frac{}{\mathbf{freeze}_{\sigma^f}([\iota_1, \iota_2;\ o;\ \rho]) \rightarrow [\iota_2;\ o;\ \rho, o \circ \sigma^f \circ \iota_1]}\quad cod(\iota_2) \cap dom(\sigma^f) = \emptyset$$

$$\text{(selection)}\quad \frac{}{[\ ;\ o;\ x_i \overset{i\in I}{\mapsto} E_i].X \rightarrow o(X)\{x_j \overset{j\in I}{\mapsto} [\ ; Y \mapsto E_j; x_i \overset{i\in I}{\mapsto} E_i].Y\}}$$

Fig. 2. Reduction rules for the untyped calculus.

For the moment we do not require meta-level substitution at the core level to satisfy any property. However, later on we will see that for defining the reduction rules (see Fig.2) and proving CR (see Sect.2) it is convenient to assume some standard properties.

Note that, with abuse of notation, we use the same symbol for meta-level substitution for both module and core terms. However this does not cause any ambiguity, since the meta-variable upon which substitution is applied clearly reveals the kind of substitution: $C\{\rho\}$ always denotes substitution at the core level, whereas $E\{\rho\}$ substitution at the module level.

Finally, note the difference between $C[x \mapsto E]$ and $C\{x \mapsto E\}$: the former denotes a particular language construct corresponding to explicit substitution, the latter is a meta-level notation for denoting the term obtained by capture avoiding substitution of the term E for the variables x in the core term C. In particular, the first expression is well-formed if E is a (well-formed) module expression, whereas the second expression is well-formed only when E is a (well-formed) core expression.

As usual, in the following we will identify any pair of terms E_1 and E_2 s.t. $E_1 \cong E_2$; more precisely, we will feel free to α-convert any term in order to avoid captures and clashes of variables while applying either substitutions or reduction steps. For instance, $[x \mapsto X;\ V \mapsto y;\ ]\{y \mapsto x\}$ is identified with the term $[z \mapsto X;\ V \mapsto x;\ ]$; indeed, $[x \mapsto X;\ V \mapsto y;\ ]\{y \mapsto x\}$ is not defined, since x would be captured. However, $[x \mapsto X;\ V \mapsto y;\ ] \cong [z \mapsto X;\ V \mapsto y;\ ]$ and $[z \mapsto X;\ V \mapsto y;\ ]\{y \mapsto x\} = [z \mapsto X;\ V \mapsto x;\ ]$.

1.2 Reduction Rules

The reduction rules for the untyped calculus are defined in Fig.2.

Moreover, we assume the usual rule for contextual closure.

$$\text{(ctx)}\quad \frac{E_1 \rightarrow E_2}{\mathbb{E}[E_1] \rightarrow \mathbb{E}[E_2]}$$

A one hole context $\mathbb{E}[\ ]$ is inductively defined by the following rules:

$$\begin{array}{l}\mathbb{E}[\,] ::= [\,] \mid C[\rho[\,]] \mid [\,\iota;\ o[\,];\ \rho] \mid [\,\iota;\ o;\ \rho[\,]] \mid \mathbb{E}[\,] + E \mid E + \mathbb{E}[\,] \mid {}_{\sigma^\iota|}\mathbb{E}[\,]_{|\sigma^o} \mid \\ \qquad\quad \mathbf{freeze}_{\sigma^f}(\mathbb{E}[\,]) \mid \mathbb{E}[\,].X \\ o[\,] ::= X \mapsto \mathbb{E}[\,], o \\ \rho[\,] ::= x \mapsto \mathbb{E}[\,], \rho\end{array}$$

By definition, the one step reduction relation $\rightarrow$ is the relation over well-formed terms inductively defined by the rules in Fig.2 plus the rule (ctx). For this reason, we have omitted all side conditions ensuring well-formedness of terms, since those are satisfied by definition.

The equational theory is convertibility, that is, equality is the convertibility relation $\leftrightarrow$ (the transitive reflexive symmetric closure of $\rightarrow$); thanks to CR (proved in Sect.2) the relation $\leftrightarrow$ can be characterized as follows: $E_1 \leftrightarrow E_2$ iff there exists E s.t. $E_i \overset{*}{\rightarrow} E$, for $i = 1, 2$ (where $\overset{*}{\rightarrow}$ denotes the transitive reflexive closure of $\rightarrow$).

Core: the definition of the reduction relation for the module calculus is parametric in the one step reduction relation $\rightarrow_C$ of the core; however, the rule uses the transitive closure of $\rightarrow_C$ (denotated by $\overset{+}{\rightarrow}_C$). This choice simplifies the proof of CR (see Sect.2), while retaining the same convertibility relation. Note that the rule can be applied only when both $C[\rho]$ and $C'[\rho]$ are well-defined, that is, ρ is well-defined and assumptions (1) and (2) of Sect.1.1 hold for both terms. However, if we require the core language to verify the following assumption, then the well-formedness of $C[\rho]$ clearly implies that of $C'[\rho]$.

Assumption 1.1
For any pair of core terms C and C', if $C \overset{+}{\rightarrow}_C C'$, then $FV_C(C') \subseteq FV_C(C)$.

Substitution: the (sub) rule applies an explicit substitution to a core term; note that, since the right hand side uses meta-level substitution at the core level, an explicit substitution can be "applied" only when a variable is substituted with a core term, that is, a term of the form $C[\rho]$. As happens for the reduction rule (core), the rule (sub) can be applied only when both $C[x \mapsto C'[\rho_1], \rho_2]$ and $C\{x \mapsto C'\}[\rho_1, \rho_2]$ are well-defined. However, if we require the core language to verify the following assumption, then the well-formedness of the term in the left-hand side clearly implies that of the term in the right-hand side (after an appropriate α-conversion of the left-hand side term, in order to have $dom(\rho_1) \cap dom(\rho_2) = \emptyset$).

Assumption 1.2
For any pair of core terms C and C' and any variable x, $FV_C(C\{x \mapsto C'\}) \subseteq (FV_C(C) \setminus \{x\}) \cup FV_C(C')$.

Sum: the reduction rule for sum is straightforward; this operation simply has the effect of gluing together two modules. However, particular attention is needed in order to correctly apply this rule.

First of all, we have to pay attention that the free variables of one module are not captured by the binders of the other (side condition). Furthermore, recall that the reduction is defined only over well-formed terms, therefore the binders of one module must be disjoint from those of the other ($BV(E_1) \cap BV(E_2) = \emptyset$). These two side conditions (both the explicit and the implicit) can always be satisfied by an appropriate α-conversion.

For the same reason of well-formedness, the output components of the two modules must be disjoint ($dom(o_1) \cap dom(o_2) = \emptyset$); 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 components in an appropriate way by means of the reduct operator (see below), thus changing the term.

The sets of the input components of the two modules can have a non empty intersection and the resulting set of the input components of the sum is simply the union of them; this means that the input components having the same name in the two modules are shared in the resulting sum.

Finally, note that sum represents a very primitive way of assembling together two modules, since it provides no way for inter-connecting their components (apart from the fact that input components are shared[2]). This can be done only at a second stage, after sum has been performed, by means of the freeze operator (see below). In other words, sum corresponds to the ability of collecting pieces of unrelated code.

In the sequel we will consider also an alternative version of *CMS* with left preferential sum $\leftarrow$ instead of the (non preferential) sum operator presented in Fig.1 and 2; its reduction rule is defined as follows:

$$\text{(left-sum)}\ \frac{}{[\iota_1;\ o_1, o;\ \rho_1] \leftarrow [\iota_2;\ o_2;\ \rho_2] \rightarrow [\iota_1, \iota_2;\ o_1, o_2;\ \rho_1, \rho_2]}\quad \begin{array}{l} BV(E_1) \cap FV(E_2) = \emptyset \\ BV(E_2) \cap FV(E_1) = \emptyset \\ dom(o) \subseteq dom(o_2) \\ E_i = [\iota_i;\ o_i;\ \rho_i],\ i = 1,2 \end{array}$$

Unlike a non-preferential sum, the expression $E_1 \leftarrow E_2$ can be reduced also when the output components in E_1 and E_2 are not disjoint (that is, when o is not empty); in this case the definitions in E_2 override the definitions of the corresponding components in E_1. As happens for the rule (sum), the implicit condition $dom(o_1) \cap dom(o_2) = \emptyset$ must be satisfied for the rule (left-sum) to be applicable.

Note that these two versions of sum cannot be expressed in terms of each other. More precisely, for fixed modules E_1 and E_2, $(E_1 \leftarrow E_2)$ can be expressed in terms of the other operators: first restrict the exports of E_1, then sum it with E_2. However, it is not possible to define an abstraction over arbitrary modules E_1 and E_2 that performs a left sum, because an abstraction cannot know the overlap between the exports of E_1 and E_2.

However, in Sect.3 we will show that in typed *CMS* the left preferential sum operator can be expressed in terms of the non preferential; more precisely, a family of left sum operators can be expressed, one for each combination of module types. Therefore, rather than considering a calculus having both operators, we prefer to keep a minimal version. In the sequel, unless differently specified, by *CMS* we mean the calculus with non preferential sum.

[2] We could avoid implicit sharing of input components in the (sum) rule by requiring $dom(\iota_1) \cap dom(\iota_2) = \emptyset$, thus forcing the user to make this sharing explicit by means of the reduct operator.

Reduct: the reduct operator performs a renaming of component names and does not change the ρ-assignment and the variables of a module; its effect is simply a composition of maps which can be correctly performed only if $cod(\iota) \subseteq dom(\sigma^\iota)$ and $cod(\sigma^o) \subseteq dom(o)$ (implicit side condition). This means that we allow a slightly more general notion of map composition (the standard one would require $cod(\iota) = dom(\sigma^\iota)$ and $cod(\sigma^o) = dom(o)$) where inclusions may be implicitly used.

This form of renaming turns out to be rather powerful: indeed, input and output names are renamed independently, by specifying two renamings σ^ι and σ^o, respectively[3]. The two renamings are contravariant for the same reason that a function from A to B can be converted into a function from A' to B' whenever two conversion functions from A' to A and from B to B' are provided.

Note that the two renamings can be non-injective and non-surjective. A non-injective map σ^ι allows sharing of input components, whereas a non-surjective one is used for adding dummy (in the sense that no variable is associated with them) input components; a non-injective map σ^o allows duplication of definitions, whereas a non-surjective map is used for hiding output components.

As a final remark, note that the syntactic representation chosen for ι-assignments is not suitable for expressing non-surjective maps, although composition of such assignments with non-surjective renamings may produce non-surjective assignments. Hence, we represent a non-surjective assignment by associating a fresh variable with each input component which is not reached in ι. For instance, the term

$$_{\{X,W\}|}[x \mapsto X;\ Y \mapsto x, Z \mapsto 1;\ ]_{|\{Y\}}$$

reduces in one step to

$$[x \mapsto X, w \mapsto W;\ Y \mapsto x;\ ]$$

where w is a fresh variable.

Freeze: as already stated, the freeze operator is essential for binding input with output components in order to accomplish inter-connection of modules. In other words, freeze corresponds to the phase, typical of any linker, of external names resolution which immediately follows the merge of the object files. However in this case the resolution is neither implicit nor exhaustive. A renaming σ^f explicitly specifies how resolution has to be performed, associating output to input components; furthermore, the domain of σ^f can be a proper subset of all input components of the module so that resolution is partial.

The effect of applying the freeze operator is that all input components that are resolved, represented by the set $cod(\iota_1)$, disappear and all the deferred variables mapped into them, represented by the set $dom(\iota_1)$, become local.

These variables are associated with the definition of the output component to which they are bound by σ^f, that is, $o(\sigma^f(\iota_1(x))$, for all $x \in dom(\iota_1)$). Recall that

[3] In the primitive calculus there exists no relationship between the names of the input and output components and the fact that these two sets of names may not be disjoint has no semantic consequence; we will consider later (Sect.4) how to encode in the calculus module systems with *virtual*, that is, both input and output, components.

the composition is well-defined if the following implicit side conditions are verified: $cod(\iota_1) \subseteq dom(\sigma^f)$ and $cod(\sigma^f) \subseteq dom(o)$.

The explicit side condition just ensures that $cod(\iota_1)$ actually contains all the input components that have to be resolved and are specified by σ^f.

The deferred variables and the input components which are not resolved (represented by $dom(\iota_2)$ and $cod(\iota_2)$, respectively) and the o-assignment are not affected.

As an example, the module expression

$$\mathbf{freeze}_{F \mapsto G}([f \mapsto F, k \mapsto K;\ G \mapsto E(f,k);\ ])$$

reduces in one step to

$$[k \mapsto K;\ G \mapsto E(f,k);\ f \mapsto E(f,k)]$$

where $E(f,k)$ denotes an expression possibly containing the free variables f and k.

Selection: finally, output components can be accessed from the outside by means of the selection operator. Selection is legal only for modules where all input components have been resolved (called *concrete* modules), hence, for all modules having an empty ι-assignment. Furthermore, the selected component X must be in $dom(o)$ (that is, must be an output component of the module); this is an implicit side condition ensuring $o(X)$ to be well-defined.

Since definitions in modules can be mutually dependent, the expression corresponding to the selected component may contain some (necessarily local) variables $\{x_i \mid i \in I\}$ which have to be replaced with their corresponding definition. Therefore, for each $j \in I$, the variable x_j is replaced with the term $[\ ; Y \mapsto E_j; x_i \overset{i \in I}{\mapsto} E_i].Y$, which clearly reduces to the definition associated with x_j in the module. Note that recursion is obtained by propagating the ρ-assignment of E in the resulting term by means of the substitution.

As an example, the module expression

$$[;\ G \mapsto g;\ k \mapsto E(k,g), g \mapsto k].G$$

reduces in one step to

$$[;\ Y \mapsto k;\ k \mapsto E(k,g), g \mapsto k].Y$$

1.3 Conservativity

We conclude this section by showing that every instantiation of *CMS* over a core calculus *CC* corresponds to a conservative extension of *CC*.

Trivially, each closed core term C is naturally embedded in the module term $C[\ ]$. Furthermore, the calculus is stratified: explicit substitutions allow core terms to contain module terms, however if a term of form $C[\rho]$ contains a module subterm that does not reduce to a core term, then $C[\rho]$ gets stuck since rule (sub) cannot be applied; these kinds of terms will be ruled out in the typed version of the calculus given in Sect.3.

On the other hand, if all module subterms E_i in $C[\rho]$ reduce to a core term (that is, a term of form $C_i[\rho_i]$), then it is easy to prove that, by iterating (sub) reduction steps, we finally obtain a core term $C'[\,]$. The further reduction of $C'[\,]$ is core calculus dependent, but clearly we expect that the semantics of each core term in the core calculus corresponds to the semantics of its embedding in *CMS*. This is ensured by the following property:

Fact 1.3 (*Conservativity 1*)
For each pair of closed core terms C_1 and C_2 we have: $C_1[\,] \rightarrow C_2[\,]$ iff $C_1 \overset{+}{\rightarrow}_c C_2$.

Proof

$\Rightarrow$: the only applicable rule is (core).
$\Leftarrow$: just apply rule (core).

$\square$

An analogous property for the static semantics is proved in Sect.3.

2 Church-Rosser Property for *CMS*

Before proving CR for *CMS* we introduce some standard definitions and properties on *Term Rewriting Systems* (TRSs) and *Combinatory Reduction Systems* (CRSs) which will be used later on. For more technical details we refer to (Klop, 1987; Klop *et al.*, 1993).

2.1 Technical Preliminaries

Definition *2.1*
An *Abstract Reduction System* (ARS) is a pair $<A, (\rightarrow_\alpha)_{\alpha\in I}>$ consisting of a set A and a sequence of binary relations $\rightarrow_\alpha$ on A, also called reduction or rewrite relations.

If for $a, b \in A$ we have $(a, b) \in \rightarrow_\alpha$ we write $a \rightarrow_\alpha b$. The reflexive, the transitive reflexive and the transitive reflexive symmetric closures of $\rightarrow_\alpha$ are written $\overset{=}{\rightarrow}_\alpha$, $\overset{*}{\rightarrow}_\alpha$, $\leftrightarrow_\alpha$, respectively. The composition $\rightarrow_\alpha \circ \rightarrow_\beta$ is defined by $a \rightarrow_\alpha \circ \rightarrow_\beta b$ if $a \rightarrow_\alpha c \rightarrow_\beta b$ for some $c \in A$. The converse relation of $\rightarrow_\alpha$ is written $\leftarrow_\alpha$.

Definition *2.2*
If $\rightarrow_\alpha$ and $\rightarrow_\beta$ are reduction relations on A, we say that they *commute* if $\overset{*}{\leftarrow}_\beta \circ \overset{*}{\rightarrow}_\alpha = \overset{*}{\rightarrow}_\alpha \circ \overset{*}{\leftarrow}_\beta$.
We say that they *strongly commute* if $\leftarrow_\beta \circ \rightarrow_\alpha = \overset{=}{\rightarrow}_\alpha \circ \overset{=}{\leftarrow}_\beta$.

Fact 2.3
If $\rightarrow_\alpha$ and $\rightarrow_\beta$ strongly commute, then they commute.

Definition *2.4*
A reduction is Church-Rosser (or confluent) if $\overset{*}{\rightarrow}$ is self-commuting.

Theorem *2.5* (*Hindley*)

Let $<A, (\rightarrow_\alpha)_{\alpha\in I}>$ be an ARS s.t. for all $\alpha, \beta \in I$, $\rightarrow_\alpha$ and $\rightarrow_\beta$ commutes. Then the union $\bigcup_{\alpha\in I} \rightarrow_\alpha$ is CR.

We refer to (Klop, 1987) and (Klop *et al.*, 1993) for the notions of Term Rewriting System and of Combinatory Reduction System, respectively.

The following definitions and theorem on TRSs apply also to CRSs (by replacing the word "variable" with "meta-variable").

Definition *2.6*
A term is *linear* if it contains no multiple occurrences of the same variable. A TRS is *left-linear* if each LHS of all its reduction rules is a linear term.

Definition *2.7*
Let $\alpha \rightarrow \beta$ and $\gamma \rightarrow \delta$ be two rewrite rules s.t. α is unifiable with a non-variable subterm of γ. Hence, there is a context $\mathbb{E}[\ ]$, a non-variable term t and a substitution ρ s.t. $\gamma = \mathbb{E}[t]$ and $\alpha\{\rho\} = t\{\rho\}$; as a consequence, the term $\gamma\{\rho\}(= \mathbb{E}[t]\{\rho\})$ can be reduced in two possible ways: $\mathbb{E}[t]\{\rho\} \rightarrow \mathbb{E}[\beta]\{\rho\}$ and $\gamma\{\rho\} \rightarrow \delta\{\rho\}$.

The pair $<\mathbb{E}[\beta]\{\rho\}, \delta\{\rho\}>$ is called a *critical pair*. If $\alpha \rightarrow \beta$ and $\gamma \rightarrow \delta$ are the same rewrite rule, then we furthermore require that α is unifiable with a *proper* non-variable subterm of $\gamma(= \alpha)$.

Definition *2.8*
A critical pair $<s, t>$ is called *convergent* if s and t have a common reduct.

Definition *2.9*
A TRS is *orthogonal* if it is left-linear and there are no critical pairs.

Theorem *2.10*
Every orthogonal TRS is CR.

Theorem *2.11*
Let C_1 and C_2 be two left-linear CRSs defined on two disjoint alphabets. Then $C_1 \cup C_2$ is CR iff C_1 and C_2 are CR.

The proof of Theorem 2.11 can be found in (van Oostrom, 1994) (Theorem 3.4.29, page 126).

2.2 Proof of CR for CMS

In order to prove CR we need some (standard) assumptions over the core calculus (in addition to assumption 1.1 and 1.2).

Assumption 2.12
In the sequel we will consider instantiations of *CMS* over core languages satisfying assumptions 1.1 and 1.2 plus the following properties:

1. the alphabets of the core and the module calculus must be disjoint;
2. for any pair of variables x_1 and x_2 and core term C, C_1 and C_2, if $x_1 \notin FV_C(C_2)$ and $x_2 \notin FV_C(C_1)$ then
$$C\{x_1 \mapsto C_1\}\{x_2 \mapsto C_2\} = C\{x_2 \mapsto C_2\}\{x_1 \mapsto C_1\};$$

3. for any pair of variables x_1 and x_2 and core term C, C_1 and C_2, if $x_2 \notin FV_C(C) \setminus \{x_1\}$ then $C\{x_1 \mapsto C_1\}\{x_2 \mapsto C_2\} = C\{x_1 \mapsto C_1\{x_2 \mapsto C_2\}\}$;
4. for any variable x and core term C, C_1 and C_2, if $C_1 \to_C C_2$ then
$$C_1\{x \mapsto C\} \to_C C_2\{x \mapsto C\};$$
5. for any variable x and core term C, C_1 and C_2, if $C_1 \to_C C_2$ then
$$C\{x \mapsto C_1\} \to_C C\{x \mapsto C_2\};$$
6. the reduction relation $\to_C$ is CR.

Assumption 2.12.1 requires that the two calculi have different operator symbols. Assumptions 2.12.2 and 2.12.3 ensure that substitution well-behaves w.r.t. composition. Assumptions 2.12.4 and 2.12.5 require the reduction relation $\to_C$ to be closed w.r.t. substitution and context application.

Lemma *2.13*
Rules (core) and (sub) strongly commute.

Proof
By induction on the minimum of the depth of the two redexes. We underline the redex in each reduction step when not coinciding with the whole term.

Basis: we distinguish the following cases:

1. both (core) and (sub) redexes have depth 0, hence they coincide with the whole term. Assuming that $C_1 \xrightarrow{+}_C C_2$, we have the following two reductions:

$$C_1[x \mapsto C[\rho_1], \rho_2] \to_{core} C_2[x \mapsto C[\rho_1], \rho_2]$$
$$C_1[x \mapsto C[\rho_1], \rho_2] \to_{sub} C_1\{x \mapsto C\}[\rho_1, \rho_2]$$

   Now clearly $C_2[x \mapsto C[\rho_1], \rho_2] \to_{sub} C_2\{x \mapsto C\}[\rho_1, \rho_2]$. Furthermore, by assumption 2.12.4 $C_1\{x \mapsto C\} \xrightarrow{+}_C C_2\{x \mapsto C\}$, therefore $C_1\{x \mapsto C\}[\rho_1, \rho_2] \to_{core} C_2\{x \mapsto C\}[\rho_1, \rho_2]$.
2. only the (sub) redex has depth 0. The most significant sub-case is represented by the following reductions, assuming that $C_1 \xrightarrow{+}_C C_2$:

$$C[x \mapsto \underline{C_1[\rho_1]}, \rho_2] \to_{core} C[x \mapsto C_2[\rho_1], \rho_2]$$
$$C[x \mapsto C[\rho_1], \rho_2] \to_{sub} C\{x \mapsto C_1\}[\rho_1, \rho_2]$$

   Now clearly $C[x \mapsto C_2[\rho_1], \rho_2] \to_{sub} C\{x \mapsto C_2\}[\rho_1, \rho_2]$. Furthermore, by assumption 2.12.5 $C\{x \mapsto C_1\} \xrightarrow{+}_C C\{x \mapsto C_2\}$, therefore $C\{x \mapsto C_1\}[\rho_1, \rho_2] \to_{core} C\{x \mapsto C_2\}[\rho_1, \rho_2]$.
   The sub-case where the (core) redex is in ρ_1 is analogous, whereas the sub-case where the (core) redex is in ρ_2 is trivial.
3. only the (core) redex has depth 0. Easy check.

Induction Step: assume that for some term E, E' and E'' we have $E \rightarrow_{core} E'$ and $E \rightarrow_{sub} E''$, where both redexes have depth greater than 0. Then we distinguish two possible cases. If the redexes are disjoint then trivially there exists E''' s.t. $E' \rightarrow_{sub} E'''$ and $E'' \rightarrow_{core} E'''$. Otherwise the redexes are nested, therefore E can be decomposed as $E = \mathbb{E}_1[E_1]$, $E_1 = \mathbb{E}_2[E_2]$ for an appropriate non empty context $\mathbb{E}_1$ and a (possibly empty) context $\mathbb{E}_2$ and terms E_1, E_2 s.t. $E_1 \rightarrow_{core} E_1'$, $E_2 \rightarrow_{sub} E_2'$ (or, equivalently, the other way round) and $E' = \mathbb{E}_1[E_1']$, $E'' = \mathbb{E}_1[\mathbb{E}_2[E_2']]$. Therefore, since $E_1 \rightarrow_{core} E_1'$ and $E_1 = \mathbb{E}_2[\underline{E_2}] \rightarrow_{sub} \mathbb{E}_2[E_2']$, by inductive hypothesis, there exists E''' s.t. $E_1' \rightarrow_{sub} E'''$, $\mathbb{E}_2[E_2'] \rightarrow_{core} E'''$. Finally, by context closure, $E' = \mathbb{E}_1[\underline{E_1'}] \rightarrow_{sub} \mathbb{E}_1[E''']$ and $E'' = \mathbb{E}_1[\underline{\mathbb{E}_2[E_2']}] \rightarrow_{core} \mathbb{E}_1[E''']$. □

Lemma *2.14*
Rule (core) strongly self-commutes.

Proof
By induction on the minimum of the depth of the two redexes.

Basis: we distinguish the following cases:

1. both redexes have depth 0, hence they coincide with the whole term. Assuming that $C_1 \stackrel{+}{\rightarrow}_C C_2$ and $C_1 \stackrel{+}{\rightarrow}_C C_3$, we have the following two reductions:

   $C_1[\rho] \rightarrow_{core} C_2[\rho]$
   $C_1[\rho] \rightarrow_{core} C_3[\rho]$

   Now, by assumption 2.12.6, there exists a core term C_4 s.t. $C_2 \stackrel{*}{\rightarrow}_C C_4$ and $C_3 \stackrel{*}{\rightarrow}_C C_4$, therefore either $C_2[\rho] \rightarrow_{core} C_4[\rho]$ or $C_2[\rho] = C_4[\rho]$, and either $C_3[\rho] \rightarrow_{core} C_4[\rho]$ or $C_3[\rho] = C_4[\rho]$.
2. only one redex has depth 0. Easy check.

Induction Step: analogous to proof of lemma 2.13. □

Lemma *2.15*
Rule (sub) strongly self-commutes.

Proof
By induction on the minimum of the depth of the two redexes.

Basis: we distinguish the following cases:

1. both redexes have depth 0, hence they coincide with the whole term. We have the following two reductions:

   $C[x_1 \mapsto C_1[\rho_1], x_2 \mapsto C_2[\rho_2], \rho_3] \rightarrow_{sub} C\{x_1 \mapsto C_1\}[\rho_1, x_2 \mapsto C_2[\rho_2], \rho_3]$
   $C[x_1 \mapsto C_1[\rho_1], x_2 \mapsto C_2[\rho_2], \rho_3] \rightarrow_{sub} C\{x_2 \mapsto C_2\}[x_1 \mapsto C_1[\rho_1], \rho_2, \rho_3]$

   Applying (sub) once more we obtain

   $C\{x_1 \mapsto C_1\}[\rho_1, x_2 \mapsto C_2[\rho_2], \rho_3] \rightarrow_{sub} C\{x_1 \mapsto C_1\}\{x_2 \mapsto C_2\}[\rho_1, \rho_2, \rho_3]$
   $C\{x_2 \mapsto C_2\}[x_1 \mapsto C_1[\rho_1], \rho_2, \rho_3] \rightarrow_{sub} C\{x_2 \mapsto C_2\}\{x_1 \mapsto C_1\}[\rho_1, \rho_2, \rho_3]$

Now since we are assuming that $C_1[\rho_1]$ and $C_2[\rho_2]$ are well-defined we have that $FV_C(C_1) \subseteq dom(\rho_1)$ and $FV_C(C_2) \subseteq dom(\rho_2)$.
Also, $C\{x_1 \mapsto C_1\}[\rho_1, x_2 \mapsto C_2[\rho_2], \rho_3]$ and $C\{x_2 \mapsto C_2\}[x_1 \mapsto C_1[\rho_1], \rho_2, \rho_3]$ are well-defined, hence $x_2 \notin dom(\rho_1)$ and $x_1 \notin dom(\rho_2)$. Therefore $x_1 \notin FV_C(C_2)$ and $x_2 \notin FV_C(C_1)$ and we can apply assumption 2.12.2.

2. only one redex has depth 0. The most significant sub-case is represented by the following reductions:

$$C[x_1 \mapsto \underline{C_1[x_2 \mapsto C_2[\rho_1], \rho_2]}, \rho_3] \rightarrow_{sub} C\{x_1 \mapsto C_1\}[x_2 \mapsto C_2[\rho_1], \rho_2, \rho_3]$$
$$C[x_1 \mapsto C_1[x_2 \mapsto C_2[\rho_1], \rho_2], \rho_3] \rightarrow_{sub} C[x_1 \mapsto C_1\{x_2 \mapsto C_2\}[\rho_1, \rho_2], \rho_3]$$

Applying (sub) once more we obtain

$$C\{x_1 \mapsto C_1\}[x_2 \mapsto C_2[\rho_1], \rho_2, \rho_3] \rightarrow_{sub} C\{x_1 \mapsto C_1\}\{x_2 \mapsto C_2\}[\rho_1, \rho_2, \rho_3]$$
$$C[x_1 \mapsto C_1\{x_2 \mapsto C_2\}[\rho_1, \rho_2], \rho_3] \rightarrow_{sub} C\{x_1 \mapsto C_1\{x_2 \mapsto C_2\}\}[\rho_1, \rho_2, \rho_3]$$

Now since we are assuming that the term $C[x_1 \mapsto C_1[x_2 \mapsto C_2[\rho_1], \rho_2], \rho_3]$ is well-formed, we have that $FV_C(C) \subseteq \{x_1\} \cup dom(\rho_3)$, whereas from the fact that the term $C\{x_1 \mapsto C_1\}[x_2 \mapsto C_2[\rho_1], \rho_2, \rho_3]$ is well-formed, we deduce $dom(\rho_3) \cap (\{x_2\} \cup dom(\rho_2)) = \emptyset$ and, hence, $x_2 \notin FV_C(C) \setminus \{x_1\}$. Therefore we conclude by assumption 2.12.3.
The sub-cases where the inner redex is in either ρ_1 or ρ_2 are trivial.

Induction Step: analogous to proof of lemma 2.13. □

Theorem *2.16*
Under Assumption 2.12, *CMS* is CR.

Proof
Let CMS_C denote the calculus consisting only of the rules (core) and (sub).

By Fact 2.3, Theorem 2.5 and Lemmas 2.13, 2.14 and 2.15 we have that CMS_C is CR.

Furthermore, let CMS_M denote the calculus consisting of all the rules of *CMS* except (core) and (sub). Trivially CMS_M is an orthogonal CRS, therefore, by Theorem 2.10, is CR.

Now, clearly *CMS* can be obtained as the disjoint union of the two left-linear CRSs CMS_C and CMS_M, therefore we can conclude by Theorem 2.11. □

3 Typed *CMS*

In this section we address the problem of defining a type system for *CMS*. As usual, we will prove that the reduction relation defined on the untyped calculus does not get stuck on well-typed terms (progress property) and preserves types (subject reduction property).

Since here we are mainly interested in type checking rather than in type inference algorithms, the terms of the typed calculus are decorated with types so that they are slightly different from those of the untyped calculus.

The types of the calculus are defined by

$\tau ::= c\tau \mid [X_i{:}\tau_i{}^{i\in I};\ X_j{:}\tau_j{}^{j\in J}]$

A type is either a core type $c\tau$ (that is, a type of the core language) or a module type $[X_i{:}\tau_i{}^{i\in I};\ X_j{:}\tau_j{}^{j\in J}]$, abbreviated by $[\Sigma^\iota;\ \Sigma^o]$. Clearly, we must assume that the core calculus is typed.

For sake of simplicity we do not introduce recursive module types and assume that core types are disjoint from module types. Recursive types are essential for typing expressions like $[;\ M \mapsto m;\ m \mapsto [;\ M \mapsto m;\ ]]$, where modules have themselves as components; we leave to further work the ability of expressing recursive module types (see the Conclusion).

Note that, according to the definition above, core types cannot be built on top of module types, hence we are forcing the core and module language to be stratified so that modules are not first-class values. See the Conclusion for a discussion about this restriction.

A module type is a pair of *signatures* Σ^ι and Σ^o. A signature is a list of pairs consisting of a component name and a type. In the following we will identify all signatures which represent the same set of pairs (that is, order and repetitions are immaterial).

Intuitively, if a module M has type $[X_i{:}\tau_i{}^{i\in I};\ X_j{:}\tau_j{}^{j\in J}]$, then $\{X_i \mid i \in I\}$ and $\{X_j \mid j \in J\}$ represent the sets of input and output components of M, respectively.

The type annotation $X_i{:}\tau_i$ says that the input (resp. output) component X_i can be correctly bound to (resp. associated with) an expression of type τ_i.

A module type is well-formed if the two signatures Σ^ι and Σ^o turn out to be two maps from component names into well-formed types. This is formalized by the judgment $\vdash [\Sigma^\iota;\ \Sigma^o]$ defined by the following rules:

$$\frac{\Vdash\Sigma^\iota,\ \Vdash\Sigma^o}{\vdash [\Sigma^\iota;\ \Sigma^o]} \qquad \frac{\vdash \tau_i \quad \forall\, i \in I}{\Vdash X_i{:}\tau_i{}^{i\in I}}\ \forall i,j \in I.X_i = X_j \Rightarrow \tau_i = \tau_j \qquad \frac{\overset{C}{\vdash} c\tau}{\vdash c\tau}$$

where $\overset{C}{\vdash} c\tau$ is the corresponding judgment for well-formed types at the core level.

The expression Σ_1, Σ_2 denotes the union of the signatures Σ_1 and Σ_2; note that this operation is not closed w.r.t. well-formed signatures. The other operators over signatures ($\cap$, $\setminus$ and $\subseteq$) which we use in the sequel are standard (and closed w.r.t. well-formed signatures).

The syntax of the typed calculus is the same as that of the untyped version, apart from basic modules where deferred and local variables are decorated with types:

$[x_i{:}\tau_i \overset{i\in I}{\mapsto} X_i;\ X_j \overset{j\in J}{\mapsto} E_j;\ x_k{:}\tau_k \overset{k\in K}{\mapsto} E_k]$

A typed basic module is well-formed if the type decoration is coherent in the sense that if $x_{i_1}{:}\tau_{i_1}$, $x_{i_2}{:}\tau_{i_2}$ and $\iota(x_{i_1}) = \iota(x_{i_2})$ then $\tau_{i_1} = \tau_{i_2}$ for any pair of deferred variables x_{i_1}, x_{i_2}.

For instance, the module

$[f{:}int \to int \mapsto F, k{:}int \mapsto K;\ G \mapsto \lambda x{:}int.k * f(x)[k \mapsto k, f \mapsto f];\ ]$

has type $[F{:}int \to int, K{:}int;\ G{:}int \to int]$.

The typing rules for the typed calculus are defined in Fig.3.

$$\text{(var)}\quad \frac{}{\Gamma \vdash x{:}\tau} \quad \Gamma(x) = \tau$$

$$\text{(core)}\quad \frac{x_i{:}c\tau_i{}^{i\in I} \overset{C}{\vdash} C{:}c\tau \qquad \Gamma \vdash E_i{:}c\tau_i \ \forall i \in I}{\Gamma \vdash C[x_i \overset{i\in I}{\mapsto} E_i]{:}c\tau}$$

$$\text{(basic)}\quad \frac{\begin{array}{c}\vdash [X_i{:}\tau_i{}^{i\in I};\ X_j{:}\tau_j{}^{j\in J}] \\ \Gamma, x_i{:}\tau_i{}^{i\in I\cup K} \vdash E_j{:}\tau_j \qquad \forall j \in J \cup K\end{array}}{\Gamma \vdash [x_i{:}\tau_i \overset{i\in I}{\mapsto} X_i;\ X_j \overset{j\in J}{\mapsto} E_j;\ x_k{:}\tau_k \overset{k\in K}{\mapsto} E_k]{:}[X_i{:}\tau_i{}^{i\in I};\ X_j{:}\tau_j{}^{j\in J}]}$$

$$\text{(sum)}\quad \frac{\vdash [\Sigma_1^\iota, \Sigma_2^\iota;\ \Sigma_1^o, \Sigma_2^o],\ \Gamma \vdash E_1{:}[\Sigma_1^\iota;\ \Sigma_1^o],\ \Gamma \vdash E_2{:}[\Sigma_2^\iota;\ \Sigma_2^o]}{\Gamma \vdash E_1 + E_2{:}[\Sigma_1^\iota, \Sigma_2^\iota;\ \Sigma_1^o, \Sigma_2^o]} \quad \Sigma_1^o \cap \Sigma_2^o = \emptyset$$

$$\text{(reduct)}\quad \frac{\Gamma \vdash E{:}[\Sigma^\iota;\ \Sigma^o]}{\Gamma \vdash {}_{\sigma^\iota|}E_{|\sigma^o}{:}[\Sigma'^\iota;\ \Sigma'^o]} \quad \begin{array}{l}\sigma^\iota{:}\,\Sigma^\iota \to \Sigma'^\iota \\ \sigma^o{:}\,\Sigma'^o \to \Sigma^o\end{array}$$

$$\text{(freeze)}\quad \frac{\Gamma \vdash E{:}[\Sigma^f, \Sigma^\iota;\ \Sigma^o]}{\Gamma \vdash \mathbf{freeze}_{\sigma^f}(E){:}[\Sigma^\iota;\ \Sigma^o]} \quad \begin{array}{l}\sigma^f{:}\,\Sigma^f \to \Sigma^o \\ \Sigma^f \cap \Sigma^\iota = \emptyset\end{array}$$

$$\text{(selection)}\quad \frac{\Gamma \vdash E{:}[\ ;\ X_i{:}\tau_i{}^{i\in I}]}{\Gamma \vdash E.X_k{:}\tau_k} \quad k \in I$$

Fig. 3. Typing rules for the typed calculus.

A context Γ is a finite (possibly empty) sequence of assignments of well-formed types to variables where variable repetition is allowed. The predicate $\Gamma(x) = \tau$ is inductively defined as follows:

- $\emptyset(x) = \tau$ is false for any variable x and type τ;
- $(\Gamma, x{:}\tau)(x') = \tau'$ iff ($x = x'$ and $\tau = \tau'$) or ($x \neq x'$ and $\Gamma(x') = \tau'$).

In the following we will use the auxiliary functions and predicates defined below.

- $dom(\emptyset) = \emptyset$ and $dom(\Gamma, x{:}\tau) = dom(\Gamma) \cup \{x\}$;
- $\Gamma \subseteq \Gamma'$ iff for all variables x, if $\Gamma(x) = \tau$, then $\Gamma'(x) = \tau$.

In the judgments we identify all contexts Γ and Γ' s.t. $\Gamma \subseteq \Gamma'$ and $\Gamma' \subseteq \Gamma$.

The (core) typing rule expresses the dependence from the core type system; the core typing judgments have form $\Gamma \overset{C}{\vdash} C{:}c\tau$, where Γ is a context containing only core types, C a core expression and $c\tau$ a core type.

The (sum) typing rule allows sharing of input components having the same name and type, whereas the side condition prevents output components to be shared. Note that the side condition $\Sigma_1^o \cap \Sigma_2^o = \emptyset$ and the premise $\vdash [\Sigma_1^\iota, \Sigma_2^\iota;\ \Sigma_1^o, \Sigma_2^o]$ are both needed to ensure that the domains of the two maps represented by Σ_1^o and Σ_2^o, respectively, are disjoint.

The typing rule for the left preferential sum is the same as the typing rule (sum) except for the absence of the side condition. However, differently to what happens for the untyped calculus, in typed *CMS* the left preferential sum can be expressed in terms of the non preferential. Indeed, the following property can be proved: for any closed terms E_1 and E_2, if $\emptyset \vdash E_i{:}[\Sigma_i^\iota;\ \Sigma_i^o]$, for $i = 1, 2$, and $\emptyset \vdash E_1 \leftarrow E_2{:}\tau$, then

$$(E_1 \leftarrow E_2) \leftrightarrow (E_{1|\Sigma_1^o \setminus \Sigma_2^o} + E_2).$$

Note the difference between the expression

$$[;\ X \mapsto x, Y \mapsto x+1;\ x \mapsto 1]_{|\{Y\}} + [;\ X \mapsto \mathit{true};]$$

and the following expression (not in *CMS*) obtained from the former by making component names and variables coincide:

$$[;\ X \mapsto 1, Y \mapsto X+1;]_{|\{Y\}} + [;\ X \mapsto \mathit{true};].$$

The former evaluates to a well-typed expression, but not the latter (an analogous counter-example can be found in (Riecke & Stone, 1999), see also Sect.6).

The side conditions having form $\sigma\colon X_i{:}\tau_i{}^{i\in I} \to X_j{:}\tau_j{}^{j\in J}$ (see typing rules (reduct) and (freeze)) ensure that the renaming σ preserves types; formally, this means that $\sigma\colon\{X_i \mid i \in I\} \to \{X_j \mid j \in J\}$ and $\sigma(X_i) = X_j \Rightarrow \tau_i = \tau_j$ for all $i \in I$, $j \in J$.

The reduction rules for the typed calculus are simply the rules of Fig.2 annotated with types.

As happens for the dynamic semantics, the static semantics of *CMS* is a conservative extension of the core static semantics.

Fact 3.1 (*Conservativity 2*)
For each core context Γ, closed core term C and core type $c\tau$ we have: $\Gamma \vdash C[\ ]{:}c\tau$ iff $\emptyset \overset{C}{\vdash} C{:}c\tau$.

Proof

$\Rightarrow$: the only applicable rule is (core).
$\Leftarrow$: just apply rule (core) . $\square$

In order to prove progress and subject reduction, we need some further (standard) assumptions over the core language.

Assumption 3.2
In the following we will consider core languages satisfying the following additional properties (besides assumptions 1.1, 1.2 and 2.12).

1. **(Substitution)** If $\Gamma_1, x{:}c\tau_2 \overset{C}{\vdash} C_1{:}c\tau_1$, $\Gamma_2 \overset{C}{\vdash} C_2{:}c\tau_2$, and $dom(\Gamma_1) \cap dom(\Gamma_2) = \emptyset$, then $\Gamma_1, \Gamma_2 \overset{C}{\vdash} C_1\{x \mapsto C_2\}{:}c\tau_1$.
2. **(Progress)** Let $\mathcal{V}_C$ be a subset of core expressions called *core values*. If $\emptyset \overset{C}{\vdash} C{:}c\tau$ then either $C \in \mathcal{V}_C$ or $C \to_C C'$, for some $C' \in \mathcal{E}_C$.
3. **(Subject Reduction)** If $C_1 \to_C C_2$ and $\Gamma \overset{C}{\vdash} C_1{:}c\tau$ then $\Gamma \overset{C}{\vdash} C_2{:}c\tau$.

The following lemmas hold for *CMS* instantiated over core languages satisfying assumptions 3.2.

Lemma *3.3* (*Weakening*)
If $\Gamma \vdash E{:}\tau$, then $\Gamma' \vdash E{:}\tau$ for any context Γ' s.t. $\Gamma \subseteq \Gamma'$.

Proof
Easy induction over the typing rules. $\square$

$$
\begin{array}{lll}
E^v ::= & C^v[\,] & \mid \text{(core value)} \\
& [\iota;\ o;\ \rho] & \text{(module value)}
\end{array}
$$

Fig. 4. Values.

Lemma *3.4* (*Substitution*)
If $\Gamma, x_i{:}\tau_i{}^{i\in I} \vdash E{:}\tau$ and $\Gamma \vdash E_i{:}\tau_i$ for all $i \in I$, then $\Gamma \vdash E\{x_i \overset{i\in I}{\mapsto} E_i\}{:}\tau$.

Proof
By induction over the height of the proof tree for $\Gamma, x_i{:}\tau_i{}^{i\in I} \vdash E{:}\tau$.

Basis: corresponds to the typing rule (var) and can be easily proved by definition of substitution.

Induction Step: corresponds to all the other typing rules. We consider only the less obvious cases (basic) and (core).

- (basic): set $E = [x_i{:}\tau_i \overset{i\in I}{\mapsto} X_i;\ X_j \overset{j\in J}{\mapsto} E_j;\ x_k{:}\tau_k \overset{k\in K}{\mapsto} E_k]$ and assume $\Gamma, x_l{:}\tau_l{}^{l\in L} \vdash E{:}\tau$. Then, by the premiss of the rule (basic), we have $\Gamma' \vdash E_j{:}\tau_j$ for all $j \in J \cup K$, with

  $\Gamma' = \Gamma, x_l{:}\tau_l{}^{l\in L}, x_i{:}\tau_i{}^{i\in I\cup K}$;

  furthermore, by hypothesis, $\Gamma \vdash E_l{:}\tau_l$ for all $l \in L$. Now let $L' \subseteq L$ be s.t.

  $\{x_l \mid l \in L'\} = \{x_l \mid l \in L\} \setminus \{x_i \mid i \in I \cup K\}$,

  so that obviously $E\{x_l \overset{l\in L}{\mapsto} E_l\} = E\{x_l \overset{l\in L'}{\mapsto} E_l\}$ and $\Gamma' = \Gamma, x_i{:}\tau_i{}^{i\in I\cup K}, x_l{:}\tau_l{}^{l\in L'}$. Then, by inductive hypothesis, $\Gamma, x_i{:}\tau_i{}^{i\in I\cup K} \vdash E_j\{x_l \overset{l\in L'}{\mapsto} E_l\}{:}\tau_j$ for all $j \in J \cup K$; therefore we can apply (basic) and conclude by definition of substitution.
- (core): set $E = C[y_j \overset{j\in J}{\mapsto} E_j]$ and assume $\Gamma, x_i{:}\tau_i{}^{i\in I} \vdash E{:}c\tau$. Then, by the premises of the (core) rule, we have $y_j{:}c\tau_j{}^{j\in J} \overset{C}{\vdash} C{:}c\tau$ and $\Gamma, x_i{:}\tau_i{}^{i\in I} \vdash E_j{:}c\tau_j$, for all $j \in J$; furthermore, by hypothesis, $\Gamma \vdash E_i{:}\tau_i$ for all $i \in I$. Therefore, by inductive hypothesis, $\Gamma \vdash E_j\{x_i \overset{i\in I}{\mapsto} E_i\}{:}c\tau_j$, for all $j \in J$, and by rule (core) we can derive $\Gamma \vdash C[y_j \overset{j\in J}{\mapsto} E_j\{x_i \overset{i\in I}{\mapsto} E_i\}]{:}c\tau$. But, by definition, $C[y_j \overset{j\in J}{\mapsto} E_j\{x_i \overset{i\in I}{\mapsto} E_i\}] = C[y_j \overset{j\in J}{\mapsto} E_j]\{x_i \overset{i\in I}{\mapsto} E_i\}$. □

In order to state the progress property, we have to define the set $\mathcal{V}_E$ of values for *CMS*, ranged over by E^v; of course, this set will be inductively defined on top of the set $\mathcal{V}_C$ of the core values, ranged over by C^v, as shown in Fig.4.

It is easy to see that $\mathcal{V}_E \subseteq \mathcal{E}_E$; recall that, when no ambiguity arises, we simply write C instead of $C[\,]$ in module expressions.

Lemma *3.5*
For each $E^v \in \mathcal{V}_E$,

- if $\emptyset \vdash E^v{:}c\tau$, then E^v is of the form $C^v[\,]$;
- if $\emptyset \vdash E^v{:}[\Sigma^\iota;\ \Sigma^o]$, then E^v is of the form $[\iota;\ o;\ \rho]$.

Proof
Easy check. □

Theorem *3.6* (*Progress*)
If $\emptyset \vdash E{:}\tau$, then either $E \in \mathcal{V}_E$ or $E \to E'$, for some $E' \in \mathcal{E}_E$.

Proof
The proof is by induction over the typing rules.

- (var): this case is vacuous since the metarule cannot be applied when $\Gamma = \emptyset$.
- (core): in this case E is of the form $C[x_i \overset{i\in I}{\mapsto} E_i]$, for some C s.t. $x_i{:}c\tau_i{}^{i\in I} \overset{C}{\vdash} C{:}c\tau$ and E_i s.t. $\Gamma \vdash E_i{:}c\tau_i$, for all $i \in I$.
If $I = \emptyset$ then by assumption 3.2.2 either $C \in \mathcal{V}_C$, and therefore $C[\,] \in \mathcal{V}_E$, or $C \to_C C'$, and therefore by reduction rule (core) $C[\,] \to C'[\,]$.
If $I \neq \emptyset$ then for an arbitrary $i \in I$ and by inductive hypothesis either $E_i \in \mathcal{V}_E$, and therefore by Lemma 3.5 E_i is of the form $C^v[\,]$ and reduction rule (sub) can be applied, or $E_i \to E_i'$ and reduction rule (core) can be applied.
- (basic): in this case trivially $E \in \mathcal{V}_E$.
- (sum): in this case E is of the form $E_1 + E_2$, for some E_1 s.t. $\Gamma \vdash E_1{:}[\Sigma_1^\iota;\ \Sigma_1^o]$, E_2 s.t. $\Gamma \vdash E_2{:}[\Sigma_2^\iota;\ \Sigma_2^o]$ and either $E_1 \notin \mathcal{V}_E$ or $E_2 \notin \mathcal{V}_E$, hence the thesis follows by inductive hypothesis and by applying context closure, or $E_1, E_2 \in \mathcal{V}_E$, hence, by lemma 3.5, E_1, E_2 are basic modules. In this case, the thesis follows by applying the reduction rule (sum); note that the explicit side conditions and the implicit side condition $BV(E_1) \cap BV(E_2) = \emptyset$ can be always satisfied by an appropriate α-conversion, while the side condition $dom(\iota_1) \cap dom(\iota_2) = \emptyset$ is implied by the side conditions of the typing rule (sum).
- (reduct): in this case E is of the form ${}_{\sigma^\iota|}E'_{|\sigma^o}$, for some E' s.t. $\Gamma \vdash E'{:}[\Sigma^\iota;\ \Sigma^o]$, $\sigma^\iota{:}\,\Sigma^\iota \to \Sigma'^\iota$, $\sigma^o{:}\,\Sigma'^o \to \Sigma^o$ and either $E' \notin \mathcal{V}_E$, hence the thesis follows by inductive hypothesis and by applying context closure, or $E' \in \mathcal{V}_E$, hence, by Lemma 3.5, $E' = [\iota;\ o^v;\ \rho^v]$. In this case, the thesis follows by applying the reduction rule (reduct); note that the implicit side conditions $cod(\iota) \subseteq dom(\sigma^\iota)$ and $cod(\sigma^o) \subseteq dom(o)$ are implied by the side conditions of the typing rule (reduct).
- (freeze): in this case E is of the form $\mathbf{freeze}_{\sigma^f}(E')$, for some E' s.t. $\Gamma \vdash E'{:}[\Sigma^f, \Sigma^\iota;\ \Sigma^o]$, $\sigma^f{:}\,\Sigma^f \to \Sigma^o$, $\Sigma^f \cap \Sigma^\iota = \emptyset$, and either $E' \notin \mathcal{V}_E$, hence the thesis follows by inductive hypothesis and by applying context closure, or $E' \in \mathcal{V}_E$, hence, by Lemma 3.5, $E' = [\iota;\ o^v;\ \rho^v]$. In this case, the thesis follows by applying the reduction rule (freeze); note that it is possible to decompose ι as ι_1, ι_2 s.t. $cod(\iota_1) = dom(\sigma^f)$, hence the explicit side condition and the implicit side conditions $cod(\iota_1) \subseteq dom(\sigma^f)$ and $cod(\sigma^f) \subseteq dom(o)$ are implied by the side conditions of the typing rule (freeze).
- (selection): in this case E is of the form $E'.X_k$, for some E' s.t.
$\Gamma \vdash E'{:}[\ ;\ X_i{:}\tau_i{}^{i\in I}]$, $k \in I$, and either $E' \notin \mathcal{V}_E$, hence the thesis follows by inductive hypothesis and by applying context closure, or $E' \in \mathcal{V}_E$, hence, by Lemma 3.5, $E' = [\ ;\ o^v;\ \rho^v]$. In this case, the thesis follows by applying the reduction rule (selection). □

Theorem *3.7* (*Subject Reduction*)
If $E \rightarrow E'$ and $\Gamma \vdash E{:}\tau$, then $\Gamma \vdash E'{:}\tau$.

Proof
The proof is by induction over the reduction rules.

- (core): in this case we have that E and E' are of the form $C[x_i \overset{i\in I}{\mapsto} E_i]$ and $C'[x_i \overset{i\in I}{\mapsto} E_i]$, respectively, and $C \overset{+}{\rightarrow}_C C'$ (otherwise the reduction rule would not be applicable); furthermore, the judgment $\Gamma \vdash E{:}c\tau$ can be derived only by applying the type rule (core), therefore we have $x_i{:}c\tau_i{}^{i\in I} \overset{C}{\vdash} C{:}c\tau$ and $\Gamma \vdash E_i{:}c\tau_i$, for all $i \in I$. Therefore, by assumption 3.2.3, $x_i{:}c\tau_i{}^{i\in I} \overset{C}{\vdash} C'{:}c\tau$ and, by rule (core), we derive $\Gamma \vdash C'[x_i \overset{i\in I}{\mapsto} E_i]{:}c\tau$.
- (sub): in this case E and E' are of the form $C[x \mapsto C'[x_i \overset{i\in I}{\mapsto} E_i], x_j \overset{j\in J}{\mapsto} E_j]$ and $C\{x \mapsto C'\}[x_k \overset{k\in I\cup J}{\mapsto} E_k]$, respectively, with $I \cap J = \emptyset$; furthermore, the judgment $\Gamma \vdash E{:}c\tau$ can be derived only by applying the type rule (core), hence, the following judgments must be valid: $x{:}c\tau_x, x_j{:}c\tau_j{}^{j\in J} \overset{C}{\vdash} C{:}c\tau$, $\Gamma \vdash C'[x_i \overset{i\in I}{\mapsto} E_i]{:}c\tau_x$ and $\Gamma \vdash E_j{:}c\tau_j$, for all $j \in J$. Similarly, from the validity of $\Gamma \vdash C'[x_i \overset{i\in I}{\mapsto} E_i]{:}c\tau_x$ we have $x_i{:}c\tau_i{}^{i\in I} \overset{C}{\vdash} C'{:}c\tau_x$ and $\Gamma \vdash E_i{:}c\tau_i$, for all $i \in I$. Note that, since E' is a well-formed term, we have that $\{x_i \mid i \in I\} \cap \{x_j \mid j \in J\} = \emptyset$, hence, by assumption 3.2.1 we have $x_k{:}c\tau_k{}^{k\in I\cup J} \overset{C}{\vdash} C\{x \mapsto C'\}{:}c\tau$. Finally, by type rule (core), using the hypotheses $x_k{:}c\tau_k{}^{k\in I\cup J} \overset{C}{\vdash} C\{x \mapsto C'\}{:}c\tau$ and $\Gamma \vdash E_k{:}c\tau_k$, for all $k \in I \cup J$, we derive $\Gamma \vdash C\{x \mapsto C'\}[x_k \overset{k\in I\cup J}{\mapsto} E_k]{:}c\tau$.
- (sum): in this case E has form $E^1 + E^2$, with

  $$E^n = [x_{i_n}{:}\tau_{i_n} \overset{i_n\in I_n}{\mapsto} X_{i_n};\ X_{j_n} \overset{j_n\in J_n}{\mapsto} E_{j_n};\ x_{k_n}{:}\tau_{k_n} \overset{k_n\in K_n}{\mapsto} E_{k_n}],$$

  for $n = 1, 2$ (assuming $I_1 \cap I_2 = \emptyset$, $J_1 \cap J_2 = \emptyset$ and $K_1 \cap K_2 = \emptyset$), whereas E' has form $[x_i{:}\tau_i \overset{i\in I_1\cup I_2}{\mapsto} X_i;\ X_j \overset{j\in J_1\cup J_2}{\mapsto} E_j;\ x_k{:}\tau_k \overset{k\in K_1\cup K_2}{\mapsto} E_k]$. Since the judgment $\Gamma \vdash E{:}[\Sigma_1^\iota, \Sigma_2^\iota;\ \Sigma_1^o, \Sigma_2^o]$ can be derived only by applying the type rule (sum), we have $\vdash [\Sigma_1^\iota, \Sigma_2^\iota;\ \Sigma_1^o, \Sigma_2^o]$, $\Gamma \vdash E^1{:}[\Sigma_1^\iota;\ \Sigma_1^o]$ and $\Gamma \vdash E^2{:}[\Sigma_2^\iota;\ \Sigma_2^o]$. Similarly, from the validity of $\Gamma \vdash E^n{:}[\Sigma_n^\iota;\ \Sigma_n^o]$, for $n = 1, 2$, since the only applicable rule is (basic), we have $\Gamma, x_i{:}\tau_i{}^{i\in I_n\cup K_n} \vdash E_j{:}\tau_j$, for all $j \in J_n \cup K_n$ and $n = 1, 2$. Furthermore, since E' is well-formed, we have that $\{x_i{}^{i\in I_1\cup K_1}\} \cap \{x_i{}^{i\in I_2\cup K_2}\} = \emptyset$, therefore, by Lemma 3.3, we deduce $\Gamma' \vdash E_j{:}\tau_j$, for all $j \in J_1 \cup J_2 \cup K_1 \cup K_2$, with $\Gamma' = \Gamma, x_i{:}\tau_i{}^{i\in I_1\cup I_2\cup K_1\cup K_2}$. From these last judgments and from $\vdash [\Sigma_1^\iota, \Sigma_2^\iota;\ \Sigma_1^o, \Sigma_2^o]$, by virtue of type rule (basic), we deduce $\Gamma \vdash E'{:}[\Sigma_1^\iota, \Sigma_2^\iota;\ \Sigma_1^o, \Sigma_2^o]$.
- (reduct): in this case E and E' have form ${}_{\sigma^\iota|}[\iota;\ o;\ \rho]_{|\sigma^o}$ and $[\sigma^\iota \circ \iota;\ o \circ \sigma^o;\ \rho]$, respectively. Since the judgment $\Gamma \vdash E{:}[\Sigma'^\iota;\ \Sigma'^o]$ can be derived only by applying the type rule (reduct), we have that $\Gamma \vdash [\iota;\ o;\ \rho]{:}[\Sigma^\iota;\ \Sigma^o]$. Similarly, from the validity of $\Gamma \vdash [\iota;\ o;\ \rho]{:}[\Sigma^\iota;\ \Sigma^o]$, since the only applicable typing rule is (basic), we have $\vdash [X_i{:}\tau_i{}^{i\in I};\ X_j{:}\tau_j{}^{j\in J}]$ and $\Gamma, x_i{:}\tau_i{}^{i\in I\cup K} \vdash E_j{:}\tau_j$ for all $j \in J \cup K$. Since $[X_i{:}\tau_i{}^{i\in I};\ X_j{:}\tau_j{}^{j\in J}]$ (corresponding to $[\Sigma^\iota;\ \Sigma^o]$) is well-formed and σ^ι and σ^o preserve types (by virtue of the side condition of the type rule (reduct)), we can easily deduce the validity of $\vdash [\Sigma'^\iota;\ \Sigma'^o]$. Furthermore, if

σ^ι is not surjective, we have $dom(\sigma^\iota \circ \iota) = dom(\iota) \cup \{x_m \mid m \in M\}$, where $\{x_m \mid m \in M\}$ is a set of fresh new dummy variables. Hence, by Lemma 3.3, we have $\Gamma, x_i{:}\tau_i{}^{i \in I \cup K}, x_m{:}\tau^{m \in M} \vdash E_j{:}\tau_j$ for all $j \in J \cup K$, where τ can be any well-formed type, and we can conclude by applying the type rule (basic).

- (freeze): in this case E and E' have form $\mathbf{freeze}_{\sigma^f}([\iota_1, \iota_2;\ o;\ \rho])$ and $[\iota_2;\ o;\ \rho, o \circ \sigma^f \circ \iota_1]$, respectively, with $cod(\iota_2) \cap dom(\sigma^f) = \emptyset$. Since the judgment $\Gamma \vdash E{:}[\Sigma^\iota;\ \Sigma^o]$ can be derived only by applying the type rule (freeze), we have that $\Gamma \vdash [\iota_1, \iota_2;\ o;\ \rho]{:}[\Sigma^f, \Sigma^\iota;\ \Sigma^o]$.
  Similarly, from the validity of $\Gamma \vdash [\iota_1, \iota_2;\ o;\ \rho]{:}[\Sigma^f, \Sigma^\iota;\ \Sigma^o]$, by applying type rule (basic), we have $\vdash [X_i{:}\tau_i{}^{i \in I};\ X_j{:}\tau_j{}^{j \in J}]$ and $\Gamma, x_i{:}\tau_i{}^{i \in I \cup K} \vdash E_j{:}\tau_j$ for all $j \in J \cup K$. Now set $I_m = dom(\iota_m)$, for $m = 1, 2$ (clearly $I_1 \cup I_2 = I$ and $I_1 \cap I_2 = \emptyset$). Since $[X_i{:}\tau_i{}^{i \in I};\ X_j{:}\tau_j{}^{j \in J}]$ is well-formed, we have that $[X_i{:}\tau_i{}^{i \in I_2};\ X_j{:}\tau_j{}^{j \in J \cup I_1}]$ is well-formed, too. Furthermore, since σ^f preserves types by the side condition of the type rule (freeze), we have that for all $i_1 \in I_1$ and $j \in J$, if $(\rho, o \circ \sigma^f \circ \iota_1)(x_{i_1}) = o(\sigma^f(X_{i_1})) = E_j$, then $\tau_{i_1} = \tau_j$, therefore we can conclude by virtue of the type rule (basic).
- (selection): in this case E and E' have form

  $[\ ;\ o;\ x_i \overset{i \in I}{\mapsto} E_i].X$ and $o(X)\{x_j \overset{j \in I}{\mapsto} [\ ; Y \mapsto E_j; x_i \overset{i \in I}{\mapsto} E_i].Y\}$,

  respectively. Since the judgment $\Gamma \vdash E{:}\tau_k$ can be derived only by applying the type rule (selection), we have that $\Gamma \vdash [\ ;\ o;\ x_i \overset{i \in I}{\mapsto} E_i]{:}[\ ;\ X_i{:}\tau_i{}^{i \in I}]$. Similarly, from the validity of $\Gamma \vdash [\ ;\ o;\ x_i \overset{i \in I}{\mapsto} E_i]{:}[\ ;\ X_i{:}\tau_i{}^{i \in I}]$, since the only applicable typing rule is (basic), we have $\vdash [\ ;\ X_i{:}\tau_i{}^{i \in I}]$ and $\Gamma, x_i{:}\tau_i{}^{i \in I} \vdash E_i{:}\tau_i$ for all $i \in I$ (in particular, we have $\Gamma, x_i{:}\tau_i{}^{i \in I} \vdash o(X){:}\tau_k$). From these judgments we can derive $\Gamma \vdash [\ ; Y \mapsto E_j; x_i \overset{i \in I}{\mapsto} E_i]{:}[\ ;\ Y{:}\tau_j]$, for all $j \in I$, by the type rule (basic), and, then, $\Gamma \vdash [\ ; Y \mapsto E_j; x_i \overset{i \in I}{\mapsto} E_i].Y{:}\tau_j$, for all $j \in I$, by the type rule (selection). Therefore, by Lemma 3.4, we conclude that $\Gamma \vdash o(X)\{x_j \overset{j \in I}{\mapsto} [\ ; Y \mapsto E_j; x_i \overset{i \in I}{\mapsto} E_i].Y\}{:}\tau_k$.
- (ctx): all cases can be easily proved by inductive hypothesis. $\square$

4 A Derived Calculus of Module Systems

The calculus *CMS* presented in the preceding sections is designed to be both powerful enough for expressing a variety of composition operators on modules, and as simple as possible. Indeed, there are only three module combinators (selection plays a different role, as explained in Sect.1.1) which correspond to three very primitive ways of manipulating modules: gluing together two modules (sum), renaming components in an arbitrary way (reduct) and binding input with output components in order to accomplish inter-connection (freeze).

While this minimal version of the calculus is the best choice from a theoretical point of view, e.g., for proving properties, the power of the calculus in terms of language design is much better illustrated by defining a derived calculus *DCMS* whose set of operators is larger and closer to constructs of concrete programming languages than that of *CMS*, but can be encoded in *CMS* in a very simple way (see

Fig.6 below). We present an abridged formal definition of *DCMS*, full details can be found in (Ancona, 1998) (Chap.4).

The calculus *DCMS* defines, on top of a generic core language, a language of mutually dependent modules allowing redefinition of components (*overriding*), also called *mixin modules* (Bracha, 1992; Ancona & Zucca, 1998b). In a language supporting mixin modules, the programmer can write open (that is, mutually depending on others) code fragments which can be type-checked and compiled separately. Moreover, it is possible to derive from existing ones new code fragments which extend and possibly modify behavior, as happens for heir classes with respect to their parents in the object-oriented approach, but in a more symmetric manner. The advantages of this approach are clear; for an extended illustration we refer to (Bracha, 1992), where the notion of mixin-based programming in the general context of module composition was introduced for the first time.

Technically speaking, there are two main novelties in *DCMS* with respect to *CMS*: first, operators behave in a *name-driven* manner (that is, binding between components is implicitly determined by equality of their names); second, components defined in a module are further divided into two classes: *virtual* and *frozen* components.

The possibility of defining virtual components is the generalization to arbitrary modular languages of a key idea of the object-oriented approach, that is, the ability of writing modules (classes) where components (methods) are simultaneously ready to be used via selection (that is, are output components), and can be modified in a way that changes the behavior of the components referring to them (that is, are input components). This is sometimes called the *open-closed* property of the object-oriented approach. We say that a module language supports *mixin modules* (or simply *mixins*) if it provides both mutual recursion and virtual components. Note that, according to this terminology, traditional object-oriented languages, even though methods of a parent and an heir class can refer to each other (hence there is mutual recursion) do not support mixins since an heir class cannot be used as a real module in the sense of the two principles mentioned in the Introduction, since it relies on a fixed parent class. Extensions of object-oriented languages with mixins (also called *mixin classes* or *parametric heir classes* in this case) are proposed in (Bracha & Griswold, 1996; Flatt *et al.*, 1998; Bono *et al.*, 1999; Ancona *et al.*, 2000).

In order to illustrate the difference between virtual and frozen components, let us introduce some syntactic sugar for *DCMS* basic modules (for simplicity we consider the untyped version):

$\texttt{import}\ X_1\ \texttt{as}\ x_1\texttt{,}\ \texttt{..,}\ X_m\ \texttt{as}\ x_m$
$\texttt{export virtual}\ V_1\ \texttt{=}\ E_1\ \texttt{as}\ v_1\texttt{,}\ \texttt{..,}\ V_n\ \texttt{=}\ E_n\ \texttt{as}\ v_n$
$\texttt{export frozen}\ F_1\ \texttt{=}\ E'_1\texttt{,}\ \texttt{..,}\ F_p\ \texttt{=}\ E'_p$
$\texttt{local}\ y_1\ \texttt{=}\ E''_1\texttt{,}\ \texttt{..,}\ y_p\ \texttt{=}\ E''_q$

As shown above, in a *DCMS* basic module the import and local parts are as in *CMS* but the exports are split in two parts, a frozen part which has the same form of the export part in *CMS* and a virtual part where a component declaration has the form $V\ \texttt{=}\ E\ \texttt{as}\ v$. This can be considered as an abbreviation for both $V\ \texttt{=}\ E$ and V as v.

As an example, consider the following concrete module `M1`, where we consider integer expressions as part of the core language:

```
module M1 is
 export virtual V = 2 as v
 export frozen F = f, Sum = v + f
 local f = 3
end M1;
```

This module, being concrete, can be effectively used; for instance the `Sum` component can be selected, obtaining 5. However, we can modify the definition of the virtual component, combining `M1` with another module `M2` defined, e.g., as follows:

```
module M2 is
 export virtual V = 4 as v
end M2;
```

by means of a left-preferential link operator, denoted by $\Leftarrow$. In the resulting module `M1`$\Leftarrow$`M2` selecting the `Sum` component would produce 7. On the contrary, if we override the `F` component by combining `M1` with `M3` defined as follows:

```
module M3 is
 export frozen F = f
 local f = 4
end M3;
```

then in the resulting module `M1`$\Leftarrow$`M3` the component `Sum` still evaluates to 5, since its definition always refers to the original value of `F`.

As shown by the example, a module with virtual components has, intuitively, two different semantics: an *open* semantics as a function, which is needed when the module is extended via overriding, and a *closed* semantics (the fixed point of the function), which is needed when the module is used via selection of a component. In (Ancona & Zucca, 1998b) we have provided a categorical denotational semantics for modules with virtual components based on this idea, originally due to (Cook, 1989; Reddy, 1988). The notion of virtual/frozen component is clearly independent of (even though inspired by) object-oriented programming and can be applied (in principle) to any module language. For a more comprehensive treatment of this subject we refer to (Bracha, 1992), where this notion was introduced for the first time, and (Ancona, 1998).

Virtual components of *DCMS* can be easily encoded in pure *CMS* by allowing a component name to appear both in the input and output assignment. For instance, the module `M1` is represented by

$$[\ v \mapsto V;\ \ V \mapsto \mathit{2}, F \mapsto f, \mathit{Sum} \mapsto v + f; f \mapsto \mathit{3}]$$

Of course, component selection in *DCMS* cannot directly be expressed by the selection operator of *CMS*, since we first need to take the closed semantics of the module; for instance, selection of *Sum* in `M1` is encoded by $\mathbf{freeze}_{V \mapsto V}(\mathit{M1}).\mathit{Sum}$.

We give now the formal definition of the derived calculus *DCMS*.

The typing rules of the language are defined in Fig.5.

Roughly speaking, a *DCMS* basic module $[\iota;\ vo;\ fo;\ \rho]$ corresponds to a *CMS*

$$\text{(basic)}\quad \frac{\vdash [\Sigma^\iota;\ \Sigma^{vo};\ \Sigma^{fo}] \qquad \Gamma, x_i{:}\tau_i{}^{i\in I\cup V\cup K} \vdash E_j{:}\tau_j \quad \forall j \in V\cup F\cup K}{\Gamma \vdash [x_i{:}\tau_i \overset{i\in I}{\mapsto} X_i; x_v{:}\tau_v \overset{v\in V}{\mapsto} X_v \overset{v\in V}{\mapsto} E_v; X_f \overset{f\in F}{\mapsto} E_f; x_k{:}\tau_k \overset{k\in K}{\mapsto} E_k]{:}[\Sigma^\iota; \Sigma^{vo}; \Sigma^{fo}]} \quad \begin{array}{l} \Sigma^\iota = X_i{:}\tau_i{}^{i\in I} \\ \Sigma^{vo} = X_v{:}\tau_v{}^{v\in V} \\ \Sigma^{fo} = X_f{:}\tau_f{}^{f\in F} \end{array}$$

$$\text{(link)}\quad \frac{\vdash [\Sigma^\iota;\ \Sigma_1^{vo}, \Sigma_2^{vo};\ \Sigma_1^{fo}, \Sigma_2^{fo}] \qquad \Gamma \vdash E_i{:}[\Sigma_i^\iota;\ \Sigma_i^{vo};\ \Sigma_i^{fo}]\ \forall i \in 1..2}{\Gamma \vdash E_1 \oplus E_2{:}[\Sigma^\iota;\ \Sigma_1^{vo}, \Sigma_2^{vo};\ \Sigma_1^{fo}, \Sigma_2^{fo}]} \quad \begin{array}{l} (\Sigma_1^{vo}, \Sigma_1^{fo}) \cap (\Sigma_2^{vo}, \Sigma_2^{fo}) = \emptyset \\ \Sigma^\iota = (\Sigma_1^\iota \setminus (\Sigma_2^{vo}, \Sigma_2^{fo})), (\Sigma_2^\iota \setminus (\Sigma_1^{vo}, \Sigma_1^{fo})) \end{array}$$

$$\text{(freeze)}\quad \frac{\Gamma \vdash E{:}[\Sigma^\iota;\ \Sigma^{vo};\ \Sigma^{fo}]}{\Gamma \vdash \mathbf{freeze}_{\Sigma^f}(E){:}[\Sigma^\iota;\ \Sigma_1^{vo};\ \Sigma_1^{fo}]} \quad \begin{array}{l} \Sigma^f \subseteq (\Sigma^{vo}, \Sigma^{fo}) \\ \Sigma_1^{vo} = \Sigma^{vo} \setminus \Sigma^f \\ \Sigma_1^{fo} = (\Sigma^{fo}, \Sigma^f) \end{array}$$

$$\text{(restrict)}\quad \frac{\Gamma \vdash E{:}[\Sigma^\iota;\ \Sigma^r, \Sigma^{vo};\ \Sigma^{fo}]}{\Gamma \vdash \mathbf{restrict}_{\Sigma^r}(E){:}[\Sigma^r, \Sigma^\iota;\ \Sigma^{vo};\ \Sigma^{fo}]} \quad \Sigma^r \cap \Sigma^{vo} = \emptyset$$

$$\text{(hide)}\quad \frac{\Gamma \vdash E{:}[\Sigma^\iota;\ \Sigma^{vo};\ \Sigma^{fo}]}{\Gamma \vdash \mathbf{hide}_{\Sigma^h}(E){:}[\Sigma^\iota;\ \Sigma_1^{vo};\ \Sigma_1^{fo}]} \quad \begin{array}{l} \Sigma^h \subseteq (\Sigma^{vo}, \Sigma^{fo}) \\ \Sigma_1^{vo} = \Sigma^{vo} \setminus \Sigma^h \\ \Sigma_1^{fo} = \Sigma^{fo} \setminus \Sigma^h \end{array}$$

$$\text{(selection)}\quad \frac{\Gamma \vdash E{:}[\ ;\ X_v{:}\tau_v{}^{v\in V};\ X_f{:}\tau_f{}^{f\in F}]}{\Gamma \vdash E \bullet X_i{:}\tau_i} \quad i \in V \cup F$$

Fig. 5. Typing rules for the *DCMS* language.

basic module $[\iota;\ o;\ \rho]$ where the o-assignment has been split into two assignments *vo* and *fo*, corresponding to the virtual and frozen components of the module, respectively. More precisely, whereas *fo* is actually an o-assignment (that is, a map from component names into expressions), *vo* is of the form $x_v{:}\tau_v \overset{v\in V}{\mapsto} X_v \overset{v\in V}{\mapsto} E_v$, where x_v is the variable used for referring to the virtual component X_v inside the module; the notation is an abbreviation for $x_v \mapsto X_v, X_v \mapsto E_v$ suggesting that virtual components are both imported and exported. The implicit conditions for well-formed terms are all inherited from *CMS* in the obvious way (in particular, note that in a well-formed assignment $x_v{:}\tau_v \overset{v\in V}{\mapsto} X_v \overset{v\in V}{\mapsto} E_v$ the names X_v are distinct for all $v \in V$).

Correspondingly, a module type is a triple $[\Sigma^\iota;\ \Sigma^{vo};\ \Sigma^{fo}]$ where Σ^{vo} and Σ^{fo} represent the virtual and frozen components, respectively. Another difference is that in *DCMS* there exists a unique name space for module components, therefore the rules for well-formed types is as follows:

$$\frac{\Vdash \Sigma^\iota, \Sigma^{vo}, \Sigma^{fo}}{\vdash [\Sigma^\iota;\ \Sigma^{vo};\ \Sigma^{fo}]} \qquad \Sigma^\iota \cap \Sigma^{vo} = \emptyset,\ \Sigma^\iota \cap \Sigma^{fo} = \emptyset,\ \Sigma^{vo} \cap \Sigma^{fo} = \emptyset$$

The link operator corresponds to that informally defined in the Introduction, where the binding between imported components of one argument and exported (either virtual or frozen) components of the other is implicitly determined by their names, with moreover the requirement that the kind (either virtual or frozen) of exported components must be preserved. Note that this operator returns a concrete module only if each imported component of E_1 is mapped into an exported component of E_2 and conversely.

Even though the link operator looks very natural as a way of assembling modules, there are few examples of concrete module languages which support this op-

$$\text{(basic)}\ \frac{\Gamma \vdash [\iota;\ vo;\ fo;\ \rho]{:}[\Sigma^\iota;\ X_v{:}\tau_v{}^{v\in V};\ \Sigma^{fo}]}{\Gamma \vdash \ll[\iota;\ vo;\ fo;\ \rho]\gg = [\iota, x_v{:}\tau_v \overset{v\in V}{\mapsto} X_v;\ X_v \overset{v\in V}{\mapsto} E_v, fo;\ \rho]}\quad vo = x_v{:}\tau_v \overset{v\in V}{\mapsto} X_v \overset{v\in V}{\mapsto} E_v$$

$$\text{(link)}\ \frac{\begin{array}{c}\Gamma \vdash E_i{:}[\Sigma^\iota_i;\ \Sigma^{vo}_i;\ \Sigma^{fo}_i]\ \forall i \in 1..2\\ \Gamma \vdash \ll E_i\gg = E'_i\ \forall i \in 1..2\\ \Gamma \vdash E_1 \oplus E_2{:}[\Sigma^\iota;\ \Sigma^{vo};\ \Sigma^{fo}]\end{array}}{\Gamma \vdash \ll E_1 \oplus E_2\gg = \mathbf{freeze}_\sigma(E'_1 + E'_2)}\quad \sigma{:}\Sigma^{fo} \cap (\Sigma^\iota_1, \Sigma^\iota_2) \hookrightarrow (\Sigma^{vo}, \Sigma^{fo})$$

$$\text{(freeze)}\ \frac{\begin{array}{c}\Gamma \vdash \ll E\gg = E'\\ \Gamma \vdash \mathbf{freeze}_{\Sigma^f}(E){:}[\Sigma^\iota;\ \Sigma^{vo};\ \Sigma^{fo}]\end{array}}{\Gamma \vdash \ll\mathbf{freeze}_{\Sigma^f}(E)\gg = \mathbf{freeze}_\sigma(E')}\quad \sigma{:}\Sigma^f \hookrightarrow (\Sigma^{vo}, \Sigma^{fo})$$

$$\text{(restrict)}\ \frac{\begin{array}{c}\Gamma \vdash E{:}[\Sigma^\iota;\ \Sigma^{vo};\ \Sigma^{fo}]\\ \Gamma \vdash \ll E\gg = E'\\ \Gamma \vdash \mathbf{restrict}_{\Sigma^r}(E){:}\tau\end{array}}{\Gamma \vdash \ll\mathbf{restrict}_{\Sigma^r}(E)\gg = E'_{|(\Sigma^{vo},\Sigma^{fo})\setminus\Sigma^r}}$$

$$\text{(hide)}\ \frac{\begin{array}{c}\Gamma \vdash E{:}[\Sigma^\iota;\ \Sigma^{vo};\ \Sigma^{fo}]\\ \Gamma \vdash \ll E\gg = E'\\ \Gamma \vdash \mathbf{hide}_{\Sigma^h}(E){:}\tau\end{array}}{\Gamma \vdash \ll\mathbf{hide}_{\Sigma^h}(E)\gg = \mathbf{freeze}_\sigma(E')_{|(\Sigma^{vo},\Sigma^{fo})\setminus\Sigma^h}}\quad \sigma{:}\Sigma^h \cap \Sigma^{vo} \hookrightarrow \Sigma^{vo}$$

$$\text{(selection)}\ \frac{\begin{array}{c}\Gamma \vdash E{:}[\ ;\ \Sigma^{vo};\ \Sigma^{fo}]\\ \Gamma \vdash \ll E\gg = E'\\ \Gamma \vdash E \bullet X{:}\tau\end{array}}{\Gamma \vdash \ll E \bullet X\gg = \mathbf{freeze}_\sigma(E').X}\quad \sigma{:}\Sigma^{vo} \hookrightarrow (\Sigma^{vo}, \Sigma^{fo})$$

Fig. 6. Translation of *DCMS* into *CMS*.

erator, allowing in practice mutually recursive definitions of modules. The proposal which more directly uses an analogous operator is that of *units* for MzScheme (Flatt & Felleisen, 1998); other proposals include Duggan and Sourelis's (Duggan & Sourelis, 1996) proposed addition to SML, the language of object files MTAL GlewMorrisett99, and the theoretical analysis in (Crary *et al.*, 1999); see Sect.7 for a comparison.

The freeze operator changes virtual into frozen components, the restrict operator changes virtual into imported components, and the hide operator hides exported (both virtual and frozen) components.

The selection operator, as in *CMS*, is used for referring to a exported (either virtual or frozen) component of a concrete module.

The semantics of *DCMS* is given by translation into *CMS* (see Fig.6). The translation needs an environment Γ since some type information must be recovered in order to give a correct translation. The notation $\Gamma \vdash \ll E\gg = E'$ means that in Γ the *DCMS* expression E is well-typed and translates into the *CMS* expression E'.

Finally, the left preferential link can be expressed in terms of the non preferential link operator, in a similar way the left preferential sum can be expressed in terms of the non preferential sum (as shown in Sect.3).

$$\frac{\Gamma \vdash E_i{:}[\Sigma^\iota_i;\ \Sigma^{vo}_i;\ \Sigma^{fo}_i]\ \forall\, i \in 1..2 \quad \Gamma \vdash \ll\!\mathbf{hide}_{\Sigma^{vo}\cup\Sigma^{fo}}(E_1) \oplus E_2\!\gg\, = E}{\Gamma \vdash \ll\! E_1 \Leftarrow E_2\!\gg\, = E} \qquad \begin{array}{l}\Sigma^{vo} = \Sigma^{vo}_1 \cap \Sigma^{vo}_2\\ \Sigma^{fo} = \Sigma^{fo}_1 \cap \Sigma^{fo}_2\end{array}$$

An extended presentation of how to translate various overriding operators, including the *super* mechanism, in a module language supporting the three basic operators of sum, reduct and freeze can be found in (Ancona & Zucca, 1997).

5 Expressive Power of the Calculus

In this section we analyze the expressive power of *CMS* by considering two translations into *CMS*, the first from the lambda-calculus (5.1), the second from the Abadi-Cardelli calculus (5.2). In particular, the first translation shows how parametric modules can be encoded in *CMS*, whereas the second sheds some light on the difference between the notion of module and object. For sake of brevity, we omit here the formal definition of the first translation which was already detailed in (Ancona & Zucca, 1999).

5.1 Parametric Modules and a Translation for the λ-calculus

Module systems like those of Standard ML (Milner *et al.*, 1990) or Objective Caml (Leroy, 2000) are based on the idea of designing the module language as a small applicative language of its own. Hence, modules are of two kinds: constant modules (*structures* in ML terminology), which can be seen in our calculus as basic modules without input components, and functions from modules into modules (*functors* in ML terminology), which can be seen in our calculus as basic modules whose input components are the expected components of the structure which is the parameter of the functor and output components are those defined by the functor itself. In these module systems, the only significant operation for composing modules is function application.

Parametric modules can be easily simulated by *CMS*-like modules, following the guideline provide by the translation from λ-calculus into *CMS* (both in the untyped version) defined in (Ancona & Zucca, 1999).

Denoting by $\ll\! e\!\gg$ the translation of a λ-term e, we have that the α- and β-rules are valid under the translation.

Proposition *5.1*
Let $\overset{\alpha,\beta}{\to}$ denotes the reduction relation over λ-terms defined by the α- and β-rules (no η-rule). Then, for each λ-term, if $e \overset{\alpha,\beta}{\to} e'$ and $\ll\! e\!\gg\, \to e''$, then $\ll\! e'\!\gg\, \leftrightarrow e''$. $FV(e_2) = FV(\ll\! e_2\!\gg)$ and by

The proof can be found in (Ancona & Zucca, 1999).

5.2 Object-Oriented Features and a Translation for the Abadi-Cardelli calculus

In the previous section we have considered languages supporting *mixins*, which, as already remarked, are not necessarily object-oriented.

$$
\begin{array}{ll}
\text{(var)} & \ll x \gg = x \\
\text{(object)} & \ll [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}] \gg = [x_i \overset{i\in 1..n}{\mapsto} \mathit{Self};\ l_i \overset{i\in 1..n}{\mapsto} \ll b_i \gg;\ ] \\
\text{(selection)} & \ll a.l \gg = \mathbf{freeze}_{\mathit{Self}\mapsto \mathit{Self}}(\ll a \gg + [;\ \mathit{Self} \mapsto \ll a \gg;\ ]).l \\
\text{(update)} & \ll a.l := \varsigma(x)b \gg = \ll a \gg \leftarrow [x \mapsto \mathit{Self}; l \mapsto \ll b \gg;\ ]
\end{array}
$$

Fig. 7. Translation of the Abadi-Cardelli object calculus into the module calculus.

For this reason, in this section we focus on a pure object-oriented language, more precisely, the Abadi-Cardelli object calculus (ACC) as defined in Chapter 6 of (Abadi & Cardelli, 1996), in order to study a formal translation from a pure object-oriented language into *CMS*.

One of the most peculiar object-oriented features not directly supported by *CMS* is the ability to refer to what is usually called the *self* object (or simply *self*), which is essential when an object is either returned as a result or passed as a parameter in the body of one of its methods.

For dealing with *self* we use the standard translation of methods into functions, see e.g., (Abadi & Cardelli, 1996), where a method with n parameters is translated into a function with $n+1$ parameters, the first corresponding to *self*. Therefore a method invocation of the form $a.m(x_1,..,x_n)$ is translated into the function application $m(a,x_1,..,x_n)$. Following this idea and recalling that functions can be easily encoded in our language (see the preceding section), we obtain the translation from ACC into *CMS* given in Fig.7.

For reasons of space, we consider only the untyped case; the translation for the typed calculus is analogous. However, note that, in the untyped case, for correctly translating the update construct we need the left preferential sum, whereas in the typed case it is possible to use also the (non preferential) sum operator, for the reasons already explained in Sect.3.

For sake of simplicity and without any loss of generality we assume that the name *Self* does not belong to the set of method names, for each expression of the form $a = [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]$ all variables x_i, for $i \in 1..n$, are distinct and the free variables[4] in a do not belong to $\{x_1,..,x_n\}$.

The following proposition shows that the equational theory of the object calculus is preserved under the translation; see (Abadi & Cardelli, 1996) for the ACC reduction rules.

Proposition *5.2*
Let $\overset{\text{acc}}{\to}$ denote the reduction relation for ACC. Then, for each ACC-term a, if $a \overset{\text{acc}}{\to} b$ and $\ll a \gg \to b'$, then $\ll b \gg \leftrightarrow b'$.

Proof
Let o denote the object $[l_i = \varsigma(x_i)b_i^{i\in 1..n}]$. Then if $k \in 1..n$ we have that $o.l_k$ reduces to $b_k[x_k \mapsto o]$. By induction on the structure of ACC-terms we can easily prove that $\ll b_k[x_k \mapsto o] \gg \equiv \ll b_k \gg [x_k \mapsto \ll o \gg]$. In the other direction, $o.l_k$ can

[4] Note that x_i may be free in b_j, for some $i,j \in 1..n$.

be directly translated into $\mathbf{freeze}_{Self \mapsto Self}(\ll o \gg + [;\ Self \mapsto \ll o \gg;\ ]).l_k$, where $\ll o \gg = [x_i \overset{i\in 1..n}{\mapsto} Self;\ l_i \mapsto \ll b_i \gg^{i\in 1..n};\ ]$.
Recalling that by assumption $Self \neq l_i$ for all $i \in I$ and applying rules (sum) and (freeze) we obtain $[;\ l_i \overset{i\in 1..n}{\mapsto} \ll b_i \gg;\ x_i \overset{i\in 1..n}{\mapsto} \ll o \gg].l_k$.
Then, by rule (selection), the term reduces to $\ll b_k \gg[x_i \overset{i\in 1..n}{\mapsto} E_i.Y]$ which is syntactically equal to $\ll b_k \gg[x_k \mapsto E_k.Y]$, by our hypothesis on the free variables of o. Since by definition $E_k = [;\ Y \mapsto \ll o \gg;\ x_i \overset{i\in 1..n}{\mapsto} \ll o \gg]$ and $E_k.Y$ reduces to $\ll o \gg[x_i \overset{i\in 1..n}{\mapsto} E_i.Y]$, which is syntactically equal to $\ll o \gg$, we can deduce that $\ll o \gg \leftrightarrow E_k.Y$ and, by congruence, $\ll b_k \gg[x_k \mapsto E_k.Y] \leftrightarrow \ll b_k \gg[x_k \mapsto \ll o \gg]$.

For object update, we have that the the term $o.l_k := \varsigma(y)b$ reduces to $[l_k = \varsigma(y)b, l_i = \varsigma(x_i)b_i^{i\in(1..n)\setminus\{k\}}]$, which is translated into $[x_i \overset{i\in 1..n}{\mapsto} Self;\ l_k \mapsto \ll b \gg, l_i \mapsto \ll b_i \gg^{i\in(1..n)\setminus\{k\}};\ ]$. In the other direction, $o.l_k := \varsigma(y)b$ can be directly translated into $\ll o \gg \leftarrow [y \mapsto Self;\ l_k \mapsto \ll b \gg;\ ]$ which reduces again to $[x_i \overset{i\in 1..n}{\mapsto} Self;\ l_k \mapsto \ll b \gg, l_i \mapsto \ll b_i \gg^{i\in(1..n)\setminus\{k\}};\ ]$. □

As further illustration of this translation, let us show a small example. Consider the object *Counter* defined by

$$Counter = [val = 0, inc = \varsigma(s)s.val := s.val + 1].$$

The method *val* returns the current value of the counter, whereas *inc* returns the counter itself where its value has been incremented by one.

The encoding of *Counter* is given by the module defined by

$$\begin{aligned} C = [&s \mapsto Self;\\ &Val \mapsto 0,\\ &Inc \mapsto (s \leftarrow [;\ Val \mapsto \mathbf{freeze}_{Self \mapsto Self}(s + [;\ Self \mapsto s;\ ]).Val + 1;\ ]);\ ] \end{aligned}$$

The deferred variable s corresponds to the ς-variable of method[5] *Inc*. The term $s \leftarrow [;\ Val \mapsto \mathbf{freeze}_{Self \mapsto Self}(s + [;\ Self \mapsto s;\ ]).Val + 1;\ ]$ is the translation of $s.val := s.val + 1$.

Since C is an open module, because of the input component *Self*, it must be closed before selecting an output component; in other words we have to use the translation rule (selection) in Fig.7. Therefore, the method invocation *Counter.inc.val* is translated into the term

$$\begin{aligned} &\mathbf{freeze}_{Self \mapsto Self}(C' + [;\ Self \mapsto C';\ ]).Val\\ &\text{with } C' = \mathbf{freeze}_{Self \mapsto Self}(C + [;\ Self \mapsto C;\ ]).Inc \end{aligned}$$

which reduces, as expected, to 1.

6 Adding Subtyping to *CMS*

At first glance it seems that the type system defined in Fig.3 can be easily improved by adding the subsumption rule, with the following definition of subtyping relation between module types:

[5] Since the method *Val* is constant we have omitted its ς-variable.

$$\frac{\begin{array}{l}\vdash [X_i{:}\tau_i{}^{i\in I};\ X_j{:}\tau_j{}^{j\in J}] \\ \vdash [X_k{:}\tau'_k{}^{k\in K};\ X_l{:}\tau'_l{}^{l\in L}] \\ \tau'_i \leq \tau_i \ \ \forall i \in I,\ \tau_l \leq \tau'_l \ \ \forall l \in L\end{array}}{[X_i{:}\tau_i{}^{i\in I};\ X_j{:}\tau_j{}^{j\in J}] \leq [X_k{:}\tau'_k{}^{k\in K};\ X_l{:}\tau'_l{}^{l\in L}]} \quad I \subseteq K, L \subseteq J$$

This definition of subtyping is standard; we can safely replace a module by another having less input and more output components; subtyping is covariant for output components and contravariant for input components.

Unfortunately, just adding the subsumption rule makes the type system unsound. To see this, consider the term $E = [;\ Y \mapsto 1;\ ] + [;\ Y \mapsto 2;\ ]$ which cannot be typed in the system of Fig.3; however, by using the subsumption rule and by definition of the subtyping relation, we have that E is correct and has type $[;\ Y{:}int]$. But the reduction of E clearly gets stuck.

This corresponds to the fact that the principle stated above "a module can be always replaced by another having less, and less specific, input and more, and more specific, output components" does not hold for some operators of the calculus, whose correct application strictly relies on the most specific types of arguments (e.g., sum).

Note that, even though in both cases this principle does not hold, this is not the same of the well-known problem of the conflict between object extension and width subtyping in extensible object languages.
We briefly recall the problem adapting the discussion in (Riecke & Stone, 1999). Assume to have an object $o = [X \mapsto 1, Y \mapsto X + 1]$, hence of type $[X{:}\mathsf{Int}, Y{:}\mathsf{Int}]$. If we allow width subtyping, then o can also have the more general type $[Y{:}\mathsf{Int}]$. At this time, there is no reason to prevent the addition of a new method X returning the value True of type Bool, thus causing a dynamic type error.

As explained in (Riecke & Stone, 1999), one possible solution to this problem is to introduce the distinction between internal and external names we have in *CMS* (see also page 21). Indeed with this distinction o becomes $[X \mapsto x, Y \mapsto x + 1, x \mapsto 1]$, and we can safely add $[X \mapsto \mathsf{True}]$ leaving Y unaffected. Hence we do not have this problem in *CMS*. On the contrary, the problem we have is not present in object languages since they usually allow to combine objects only by overriding, that is, preferential sum.

For safely introducing subtyping we have two possibilities. The first is to choose a set of basic operators which support subsumption (for instance replacing non preferential sum by preferential sum). The second is to keep the basic operators of *CMS*, and, rather than just adding a subsumption rule, to define an alternative type system where, whenever it is possible, we relax the constraints on the argument types. A proposal for the second alternative is given in Fig.8 (the other rules have been omitted, since they are unchanged).

Note that with the new rules it is enough to consider only surjective morphisms, since the non-surjective are obtained "for free" by subsumption. Moreover, note that the type system with subtyping actually is an improvement of that in Fig.3, since now it is possible to type some intuitively correct terms which were not statically correct before.

As an example, consider the term $E = \mathbf{freeze}_{\{M \mapsto N\}}(E_1 + E_2).Y$ where

$$\text{(reduct)} \quad \frac{\Gamma \vdash E{:}\tau,\ \tau \leq [\Sigma^\iota;\ \Sigma^o]}{\Gamma \vdash {}_{\sigma^\iota|}E_{|\sigma^o}{:}[\Sigma'^\iota;\ \Sigma'^o]} \quad \begin{array}{l}\sigma^\iota{:}\Sigma^\iota \to \Sigma'^\iota \\ \sigma^o{:}\Sigma'^o \to \Sigma^o\end{array}$$

$$\text{(freeze)} \quad \frac{\begin{array}{c}\Gamma \vdash E{:}[\Sigma^f, \Sigma^\iota;\ \Sigma^o] \\ [\Sigma^f, \Sigma^\iota;\ \Sigma^o] \leq [\Sigma'^f, \Sigma^\iota;\ \Sigma'^o]\end{array}}{\Gamma \vdash \mathbf{freeze}_{\sigma^f}(E){:}[\Sigma^\iota;\ \Sigma^o]} \quad \begin{array}{l}\sigma^f{:}\Sigma'^f \to \Sigma'^o \\ \Sigma'^f \cap \Sigma^\iota = \emptyset\end{array}$$

Fig. 8. Typing rules with subtyping.

$E_1 = [m{:}[X{:}int;\ Y{:}int] \mapsto M;\ Y \mapsto \mathbf{freeze}_{\{X \mapsto Y\}}(m).Y;\ ]$,
$E_2 = [;\ N \mapsto [;\ Y \mapsto 3;\ ];\ ]$.

It is easy to check that E cannot be typed with the rules of Fig.3 even though it correctly reduces to 3. Indeed,

$$\emptyset \vdash E_1 + E_2{:}[M{:}[X{:}int;\ Y{:}int];\ Y{:}int, N{:}[;\ Y{:}int]]$$

and the typing rule (freeze) cannot be applied since the type of the input component M is not equal to the type of the output component N. On the other hand we have

$$[M{:}[X{:}int;\ Y{:}int];\ Y{:}int, N{:}[;\ Y{:}int]] \leq [M{:}[;\ Y{:}int];\ N{:}[;\ Y{:}int]],$$

hence, by the new typing rule for freeze,

$$\emptyset \vdash \mathbf{freeze}_{\{M \mapsto N\}}(E_1 + E_2){:}[;\ Y{:}int, N{:}[;\ Y{:}int]]$$

and therefore we can conclude $\emptyset \vdash E{:}int$.

7 Related Work

The two authors who have more directly inspired the research presented in this paper are probably G. Bracha and X. Leroy. They have both strongly supported and clarified with their work the two principles for module systems we have mentioned in the Introduction, in particular the idea that the module language should be a small language in itself with its own typing rules constructed on top of an (in principle) arbitrary core language. Most of the module operators presented in Sect.4 come from Bracha's work (Bracha & Cook, 1990; Bracha, 1992; Bracha & Lindstrom, 1992), as well as the idea of extending to a general context of module composition the overriding mechanism of the object-oriented approach. Furthermore, (Bracha, 1992) represents the first attempt, which has been fully exploited in our own work (here and (Ancona & Zucca, 1998b; Ancona & Zucca, 1999)), to reduce many different module composition mechanism to a set of primitive operators with clean semantics. Leroy's work (Leroy, 1994; Leroy, 2000) has inspired the great importance we have attributed to achieve the real independence of our module calculus from the underlying core language and the effectiveness of the construction, in the sense explained in Sect.1.1 (a type-checker/interpreter for *CMS* can be constructed in a modular way on top of the type-checker/interpreter for the core level, as done in (Leroy, 2000)).

Advanced modular languages A large literature concerns ML-style modules (see, e.g., (Leroy, 1994; Harper & Lillibridge, 1994; Russo, 1998). In particular, as mentioned in Sect.4, (Duggan & Sourelis, 1996; Duggan & Sourelis, 1998) contain proposals for extending Standard ML with mutually recursive modules[6] and the type-theoretical analysis in (Crary *et al.*, 1999) deals with the problem of recursive type definitions spanning module boundaries in the context of the phase distinction formalism (Harper *et al.*, 1990).

Other relevant work concerns designing languages with *mixin modules* as defined in this paper, that is module languages allowing mutual recursion and overriding with late binding. We cite again (Bracha, 1992; Bracha & Lindstrom, 1992) and the subsequent work in (Banavar, 1995; Banavar & Lindstrom, 1996; Limberghen & Mens, 1996); more recently, we mention the methodological paper (Findler & Flatt, 1998), the calculus presented in (Bono *et al.*, 1999) and the proposals for mixin-based extensions of Java in (Flatt *et al.*, 1998; Ancona *et al.*, 2000).

Finally, a proposal which looks similar to our calculus is that of *units* for MzScheme (Flatt & Felleisen, 1998). Basic units are very close to basic modules of *CMS* since they are, in their graphical representation, boxes with an import, an export and an internal section (however, unlike our modules, units are run-time entities with an initialization part). Many units can be composed by a linking process which is graphically described by putting all the boxes inside a collecting box and connecting some input to export ports by arrows. This corresponds in our formalism to a composition of *link* operators plus a *reduct* operation which performs the connections from/to ports of the collecting box. Indeed, there is a natural graphical representation of all our operators over modules which closely resembles that given in (Flatt & Felleisen, 1998) for units; the interested reader can refer to (Ancona & Zucca, 1998a).

Calculi with linking In (Cardelli, 1997) Cardelli proposes a calculus of compilation units for the simply-typed lambda calculus and presents a set of rules for determining link compatibility. In (Glew & Morrisett, 1999), building upon these ideas and module constructs from high-level languages, the authors present a formal model of *typed object files* and a set of inference rules that are sufficient to guarantee that type safety is preserved by the linking process. The resulting language MTAL is similar to units (Flatt & Felleisen, 1998) and *CMS*, but is different from our approach for the following reasons:

- even though MTAL is independent of the core language, this independency is not so abstracted as in *CMS*. For instance, the syntax rules for MTAL are explicitly based on those for the core language;
- MTAL is a language of object files which deals with low level implementation issues at the basis of the linking process, whereas *CMS* is a calculus (and not

[6] Duggan and Sourelis use the name *mixins* for their mutually recursive modules; we prefer to reserve this name for modules which support both mutual recursion and overriding with dynamic binding as in the object-oriented approach (see the end of Sect.4).

only a language) where more emphasis is given to semantic aspects and proofs of general properties while implementation issues are only barely addressed;
- *CMS* provides a richer set of operations for combining modules. For instance, the reduct operator of *CMS* includes the restriction operator advocated in (Riecke & Stone, 1999) (see the final part of Sect.4.1) for alleviating the software engineering problems of a flat name space;
- MTAL deals with type components (see the comments in the Conclusion on how to introduce type components in CMS).

In (Drossopoulou *et al.*, 1999) the authors propose a calculus describing compilation and linking in terms of operation on *fragments*, that is, compilation units. More precisely, a *fragment system* consists of two sets $\mathcal{S}$ and $\mathcal{B}$ of source and binary fragments, respectively, a compilation function from $\mathcal{S} \times \mathcal{B}$ into $\mathcal{B}$, two linking operators over source and binary fragments, respectively, and two predicates expressing well-formedness of binary fragments w.r.t. to source and binary fragments, respectively. The framework is used for formally expressing the notion of binary compatibility in Java.

Finally, the two recent papers (Wells & Vestergaard, 2000) and (Machkasova & Turbak, 2000) present module calculi which are very close to *CMS*. In both proposals the basic underlying ideas are similar to ours, but much more emphasis is given to equational reasoning.

In the m-calculus presented in (Wells & Vestergaard, 2000), a basic module has the form (using our meta-variables)

$$X_1 \triangleright x_1 = E_1, .., X_n \triangleright x_n = E_n$$

that is, is a collection of components where in each $X_i \triangleright x_i = E_i$ either the X_i or the E_i can be missing; in the former case the component is called *private*, in the latter *deferred* or *input*, while a component having both X_i and E_i is an *exported* or *output* component. As it should be clear, a basic module of the m-calculus can be immediately converted in a *CMS* basic module as follows:

- every input component $X_i \triangleright x_i = \bullet$ is translated into a *CMS* input component $x_i \mapsto X_i$,
- every output component $X_i \triangleright x_i = E_i$ is translated into a *CMS* output component $X_i \mapsto x_i$, plus a *CMS* local component $x_i \mapsto E_i$,
- every local component $\_ \triangleright x_i = E_i$ is translated into a *CMS* local component $x_i \mapsto E_i$.

Operators of the m-calculus all behave in a name-driven way, so the comparison actually is more directly made with the *DCMS* calculus presented in Sect.4. Indeed the translation above expresses the fact that a basic m-calculus is a basic *DCMS* module without virtual components (all components are implicitly frozen in the m-calculus). More precisely, the m-calculus provides the following operators:

- a link operator corresponding to the link of *DCMS* in the case with no virtual components;

- a component hiding operator corresponding to the *DCMS* hiding or, equivalently, restrict (there is no difference since the m-calculus has no virtual components);
- a component sieving operator which is complementary to hiding; this operator is necessary in the m-calculus since it cannot be expressed in terms of the others; it cannot be translated into *DCMS*, but can be directly translated into *CMS* by means of the reduct operator;
- component selection, corresponding to *DCMS* or, equivalently, *CMS* component selection (there is no difference since the m-calculus has no virtual components);
- a *letrec* construct (a pair $\langle M|D\rangle$ consisting of a module plus a collection of private components) which is basically a different technical solution for representing terms resulting in the reduction of selection in presence of mutual recursion.

Finally, the m-calculus is not parametric in the core language, but constructed on just variables.

While the syntax of the m-calculus is equivalent to a subset of *DCMS* (except for the sieving operator), the reduction rules are very different. Indeed, the semantics of *CMS* has been designed as a "pure" reduction semantics, in the sense that all the reduction rules correspond to a simplification step which eliminates a composition operator (gluing together two basic modules, or performing a renaming, or performing a connection between input and output components). Thus, the equivalence induced by our reduction rules does not capture equalities that hold in the m-calculus; for instance, $x + y = y + x$ (commutativity of sum of module variables) or $[;\ V \mapsto 1;\ x \mapsto 0] = [;\ V \mapsto 1;\ ]$ (useless local variables). In (Wells & Vestergaard, 2000) a considerable effort is devoted to obtaining these equivalences by adding appropriate reduction rules. As a consequence, this choice complicates the semantics and the proof of confluence, but the advantage is that it is possible to get most of the expected equalities between module expressions "for free", whereas in *CMS* they must be proved in a possibly involved way. In summary, the emphasis in (Wells & Vestergaard, 2000) is much more on equational reasoning, while typing aspects and parametricity w.r.t. the core language are not investigated.

The module calculus in (Machkasova & Turbak, 2000) serves as a framework for *link-time compilation* (a model of compilation that lies in the expanse between *whole-program compilation* and *separate compilation*). The calculus is stratified into three levels. The first level is the core calculus (following our terminology), which is assumed here to be call-by-value lambda calculus (as in (Wells & Vestergaard, 2000), parametricity w.r.t. the core level is not considered). The second level, called the *core module calculus*, consists of basic modules, which are of the form

$$[l_1 \mapsto C_1, .., l_n \mapsto C_n]$$

where $l_1, .., l_n$ are *labels* and $C_1, .., C_n$ are expressions of the first level, that is, core expressions (the calculus does not support nested modules, i.e. modules as components). Labels are distinct from variables and correspond to our component names, even though they can be either *visible* or *hidden*. Hidden labels play the role of our local variables: indeed in (Machkasova & Turbak, 2000) labels can be directly used inside (core) expressions exactly as variables. There are three reduction rules for the core module calculus: propagation of a core step, substitution of a label with the associated expression and elimination of useless labels. The third level, called the *full module calculus*, introduces two module combinators (link, which corresponds to the sum of *CMS*, and renaming of labels) with the corresponding reduction rules.

As in (Wells & Vestergaard, 2000) and differently from us, the authors provide more module level reduction rules than just those corresponding to elimination of module combinators. However, differently from both (Wells & Vestergaard, 2000) and us, their calculus does not include an explicit selection operator, which is not necessary in the context of link-time compilation. Another remarkable difference is that the calculus in (Machkasova & Turbak, 2000) is not confluent; indeed, the authors show an example, inspired by (Ariola & Klop, 1997), where application of the substitution reduction rule to mutually dependent module components leads to two different terms that can never meet at a common term. This problem does not arise in *CMS* simply because there is no substitution rule, whereas in (Wells & Vestergaard, 2000) the problem is circumvented by allowing substitution only in the absence of mutual dependency. However, the authors show that a restricted form of *confluence with respect to evaluation* holds which is enough for establishing the Observational Equivalence Property (if two terms are equivalent in the calculus, then they are observationally equivalent).

8 Conclusion

We have presented *CMS*, a simple and powerful calculus for module systems equipped with a confluent reduction semantics and a type system enjoying progress and subject reduction. Moreover, we have illustrated that it can be actually used as a primitive kernel in which to encode various existing mechanisms for combining software components. We have also implemented an interpreter for the untyped *CMS*, parametric in the interpreter of the core calculus[7]. As already pointed out in the Introduction, this parametric interpreter allows the user to define core calculi where modules are first class values, thus breaking the stratification imposed by the formalism defined here, which, therefore, turns out to be less flexible than the proposed implementation. We leave to further investigation the possibility of defining a parametric framework able to express mutual dependency between the two calculi.

A point which deserves discussion is the treatment of type components in modules, hence mutually recursive type definitions that span module boundaries, as one would have, e.g., instantiating module operators of *CMS* on a ML-like core language.

[7] See `http://www.disi.unige.it/person/AnconaD/Java/UPCMS.html`.

The primitive calculus *CMS* presented in this paper does not have a distinguished notion of type components since we are mainly interested in defining a set of both powerful and simple primitive module operators for the manipulation of open fragments, which could be source fragments combined by operators at the language level as well as code fragments combined by a generalization of a traditional linker. In particular, *CMS* could be safely instantiated on a core language of type expressions. However, the instantiation on a core language supporting a two level type system (type expressions having kinds and value expressions having types) requires some refinement of the framework. A standard approach based on the notion of *phase distinction* (see (Harper *et al.*, 1990)) has been adopted in (Glew & Morrisett, 1999) and indeed can be applied to *CMS* as well; in this case, a *CMS* basic module would consist of a pair $<M_1, M_2>$ of modules, where M_1 is a basic module of type components independent of M_2, whereas M_2 is a basic module of value components depending on M_1. As a consequence, the *CMS* module operators need to be extended.

In (Ancona & Zucca, 1998b), where we have provided a denotational (categorical) semantics, parametric in the core level, for the three primitive operators (excluding selection) defined in this paper, we have also already formally defined a refined version of the operators which treats appropriately type components. Defining the analogue of *CMS*, that is, a true calculus equipped with a reduction semantics, for this refined version has still to be done and is an important subject of further work; we expect more additional ingredients than those required in this paper to be needed at the core level, e.g., a syntax for type definitions, or for *type constraints* (Ancona, 1999) if we want to take into account a more flexible approach allowing types to be "partially" specified.

In the direction of the application to concrete languages, we have already applied the module language presented in this paper to core languages supporting type definitions; notably, to a simple functional language with sum and product types in (Ancona, 2000) and to a Java-like language with class types in (Ancona & Zucca, 2001). Moreover, an extension of Caml with mixin modules based on *CMS* is under development (Hirschowitz, 2000).

Still concerning type aspects, another issue for further work is the introduction of more complicated type systems, e.g., including recursive module types and/or polymorphism.

Another interesting research direction is that related to the definition of an operational semantics (hence an observational equivalence) and the study of further properties of the calculus: for instance, the Observational Equivalence Property (if two terms are equivalent in the calculus, then they are observationally equivalent). of non-standard As mentioned in Sect.7, proving the Observational Equivalence Property could imply some effort that in other similar calculi (Wells & Vestergaard, 2000; Machkasova & Turbak, 2000) is, in some sense, avoided by directly including some equivalences at the level of the reduction relation.

Finally, an important topic we would like to study in the future is how to express in the framework of module calculi properties which involve different module languages, as typically happens in compilation, where source modules and binary modules are both involved. This is especially relevant in languages like Java, where bytecode modules retain relevant type information in order to make possible dynamic loading of classes, hence the combination of modules at the binary level is not trivial; also, to express safe code transformations in compilers. As mentioned in Sect.7, some preliminary steps in this direction are in (Drossopoulou *et al.*, 1999; Machkasova & Turbak, 2000). However, we think that much work remains to be done on this subject.

Acknowledgments

We warmly thank Xavier Leroy and Eugenio Moggi for many useful suggestions on a preliminary version of this calculus. In particular, Xavier provided interesting hints about the encoding of λ-calculus during his visit in Genova in September 1998, and Eugenio Moggi strongly encouraged and helped us to adopt the current formulation based on explicit substitutions.

We are grateful to Lyn Turbak and Joe Wells for the interesting discussions about the topics of this paper. Concerning the proof of CR, we are in debt with Jan Willem Klop, whose help avoided us many (other) troubles.

Finally, many thanks also to Zino Benaissa, Mark Jones and Walid Taha, and to the anonymous referees for their careful reading and the number of corrections and suggestions which provided invaluable help in improving this paper.

References

Abadi, M., & Cardelli, L. (1996). *A theory of objects.* Monographs in Computer Science. Springer.

Ancona, D. (1998). *Modular formal frameworks for module systems.* Ph.D. thesis, Dipartimento di Informatica, Università di Pisa.

Ancona, D. (1999). An algebraic framework for separate type-checking. *Pages 1–15 of:* Fiadeiro, J. (ed), *WADT'98 (13th Workshop on Algebraic Development Techniques).* Lecture Notes in Computer Science, no. 1589. Springer.

Ancona, D. (2000). MIX(FL): a kernel language of mixin modules. Rus, T. (ed), *AMAST 2000 - International Conference on Algebraic Methodology and Software Technology.* Lecture Notes in Computer Science, no. 1816. Springer.

Ancona, D., & Zucca, E. (1997). Overriding operators in a mixin-based framework. *Pages 47–61 of:* Glaser, H., Hartel, P., & Kuchen, H. (eds), *PLILP '97 - 9th Intl. Symp. on Programming Languages, Implementations, Logics and Programs.* Lecture Notes in Computer Science, no. 1292. Springer.

Ancona, D., & Zucca, E. (1998a). An algebra of mixin modules. *Pages 92–106 of:* Presicce, F. Parisi (ed), *Recent Trends in Algebraic Development Techniques (12th Intl. Workshop, WADT'97 - Selected Papers).* Lecture Notes in Computer Science, no. 1376. Springer.

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

Ancona, D., & Zucca, E. (1999). A primitive calculus for module systems. *Pages 62–79 of:* Nadathur, G. (ed), *PPDP'99 - Principles and Practice of Declarative Programming.* Lecture Notes in Computer Science, no. 1702. Springer.

Ancona, D., & Zucca, E. (2001). True modules for Java classes. *Pages 354–380 of:* Knudsen, J.L. (ed),*ECOOP 2001 - Object-Oriented Programming.* Lecture Notes in Computer Science, no. 2072. Springer.

Ancona, D., Lagorio, G., & Zucca, E. (2000). Jam: A smooth extension of Java with mixins. *Pages 154–178 of:* Bertino, E. (ed), *ECOOP'00 - Object-Oriented Programming.* Lecture Notes in Computer Science, no. 1850. Springer.

Ariola, Z. M., & Klop, J. W. (1997). Lambda calculus with explicit recursion. *Information and Computation*, **139**(2), 154–233.

Banavar, G. (1995). *An application framework for compositional modularity.* Ph.D. thesis, Department of Comp. Sci., Univ. of Utah.

Banavar, G., & Lindstrom, G. (1996). An application framework for module composition tools. *Pages 91–113 of: ECOOP '96 - Object-Oriented Programming.* Lecture Notes in Computer Science, no. 1098. Springer.

Bono, V., Patel, A., & Shmatikov, V. (1999). A core calculus of classes and mixins. *Pages 43–66 of:* Guerraoui, R. (ed), *ECOOP '99 - Object-Oriented Programming.* Lecture Notes in Computer Science, no. 1628. Springer.

Bracha, G. (1992). *The programming language JIGSAW: Mixins, modularity and multiple inheritance.* Ph.D. thesis, Department of Comp. Sci., Univ. of Utah.

Bracha, G., & Cook, W. (1990). Mixin-based inheritance. *Pages 303–311 of: ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1990.* ACM Press. SIGPLAN Notices, volume 25, number 10.

Bracha, G., & Griswold, D. 1996 (April). Extending Smalltalk with mixins. *OOPSLA96 Workshop on Extending the Smalltalk Language.* Electronic note available at `http://www.javasoft.com/people/gbracha/mwp.html`.

Bracha, G., & Lindstrom, G. (1992). Modularity meets inheritance. *Pages 282–290 of:*

Proc. International Conference on Computer Languages. San Francisco: IEEE Computer Society.

Cardelli, L. (1997). Program fragments, linking, and modularization. *Pages 266–277 of: ACM Symp. on Principles of Programming Languages 1997*. ACM Press.

Cook, W. (1989). *A denotational semantics of inheritance*. Ph.D. thesis, Dept. Comp. Sci., Brown University.

Crary, K., Harper, R., & Puri, S. (1999). What is a recursive module? *PLDI'99 - ACM Conf. on Programming Language Design and Implementation*.

Curien, P.-L., Hardin, T., & Levy, J.-J. (1996). Weak and strong confluent calculi of explicit substitutions. *Journ. ACM*, **43**(2).

Duggan, D., & Sourelis, C. (1996). Mixin modules. *Pages 262–273 of: Intl. Conf. on Functional Programming*. ACM Press.

Duggan, D., & Sourelis, C. (1998). Parameterized modules, recursive modules, and mixin modules. *Pages 87–96 of: 1998 ACM SIGPLAN Workshop on ML*. ACM Press.

Findler, R.B., & Flatt, M. 1998 (September). Modular object-oriented programming with units and mixins. *Intl. Conf. on Functional Programming 1998*.

Flatt, M., & Felleisen, M. (1998). Units: Cool modules for HOT languages. *Pages 236–248 of: PLDI'98 - ACM Conf. on Programming Language Design and Implementation*.

Flatt, M., Krishnamurthi, S., & Felleisen, M. 1998 (January). Classes and mixins. *Pages 171–183 of: ACM Symp. on Principles of Programming Languages 1998*. ACM Press.

Glew, N., & Morrisett, J.G. (1999). Type-safe linking and modular assembly language. *Pages 250–261 of: ACM Symp. on Principles of Programming Languages 1999*. ACM Press.

Harper, R., & Lillibridge, M. (1994). A type theoretic approach to higher-order modules with sharing. *Pages 127–137 of: ACM Symp. on Principles of Programming Languages 1994*. ACM Press.

Harper, R., Lillibridge, M., & Moggi, E. (1990). Higher-order modules and the phase distinction. *Pages 341–354 of: ACM Symp. on Principles of Programming Languages 1990*. ACM Press.

Hirschowitz, T. (2000). Modules mixins : typage et compilation. Rapport du DEA Programmation: Sémantique, Preuves et Langages. INRIA Rocquencourt. `http://cristal.inria.fr/ hirschow/`.

Klop, J.W. (1987). Term rewriting systems: a tutorial. *Bull. of EATCS*, **32**, 143–182.

Klop, J.W., van Oostrom, V., & van Raamsdonk, F. (1993). *Combinatory reduction systems: introduction and survey*. Tech. rept. CS-R9362. CWI.

Leroy, X. (1994). Manifest types, modules and separate compilation. *Pages 109–122 of: ACM Symp. on Principles of Programming Languages 1994*. ACM Press.

Leroy, X. (2000). A modular module system. *Journal of functional programming*, **10**(3), 269–303.

Limberghen, M. Van, & Mens, T. (1996). Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, **3**(1), 1–30.

Machkasova, E., & Turbak, F.A. (2000). A calculus for link-time compilation. *Pages 260–274 of: European Symposium on Programming 2000*. Lecture Notes in Computer Science, no. 1782. Springer.

Milner, R., Tofte, M., & Harper, R. (1990). *The definition of Standard ML*. The MIT Press.

Reddy, U. S. (1988). Objects as closures: Abstract semantics of object-oriented languages. *Pages 289–297 of: Proc. acm Conf. on Lisp and Functional Programming*.

Riecke, J.G., & Stone, C.A. (1999). Privacy via subsumption. *Theory and Practice of Object Systems*. To appear.

Russo, C.V. (1998). *Types for modules*. Ph.D. thesis, Dept. Comp. Sci., Univ. Edinburgh.

Drossopoulou, S. & Eisenbach, S., & Wragg, D. 1999 (July). A fragment calculus - towards a model of separate compilation, linking and binary compatibility. *Proc. 14th Ann. IEEE Symp. on Logic in Computer Science*.

van Oostrom, V. (1994). *Confluence for abstract and higher-order rewriting*. Ph.D. thesis, VU, Amsterdam.

Wells, J.B., & Vestergaard, R. (2000). Confluent equational reasoning for linking with first-class primitive modules. *Pages 412–428 of: European Symposium on Programming 2000*. Lecture Notes in Computer Science, no. 1782. Springer.