Equational Reasoning for Linking with First-Class Primitive Modules

J. B. Wells[1]* and René Vestergaard[1]

Heriot-Watt University

Abstract. Modules and linking are usually formalized by encodings which use the λ-calculus, records (possibly dependent), and possibly some construct for recursion. In contrast, we introduce the m-calculus, a calculus where the primitive constructs are modules, linking, and the selection and hiding of module components. The m-calculus supports smooth encodings of software structuring tools such as functions (λ-calculus), records, objects (ς-calculus), and mutually recursive definitions. The m-calculus can also express widely varying kinds of module systems as used in languages like C, Haskell, and ML. We prove the m-calculus is confluent, thereby showing that equational reasoning via the m-calculus is sensible and well behaved.

1 Introduction

A long version of this paper [44] which contains full proofs, more details and explanations, and comparisons with more calculi (including the calculus of Ancona and Zucca [5]), is available at `http://www.cee.hw.ac.uk/~jbw/papers/`.

1.1 Support for Modules in Established Languages

All programming languages need support for modular programs. For languages like C, conventions outside the definition of the language provide this support. Each source file is compiled to an object ("`.o`") file which plays the role of the module. The namespace of modules is simply the file system and linking of modules is specified via extra-linguistic mechanisms such as makefiles. Connections are hard-wired to the component name rather than the module name: If module X uses module Y, modules Z and W supplying components with the same names as those of Y can be substituted for Y. There is a single global namespace for component names. Mutual dependencies between modules is possible, but there is no mechanism for black-box reuse of modules and no support for hierarchical structuring of modules within modules.

Languages like Ada [10], Modula-3 [26], and Haskell [1] support a kind of module which we will call *packages*. With packages, there is a flat namespace of modules; by convention module names correspond to filenames. Connections are hard-wired to module names: If module X uses module Y, then any replacement

* Supporting grants: EPSRC GR/L 36963, NSF CCR–9417382 and CCR–9806747.

for Y must also be named Y and support at least the components used by X. As with C, mutual dependencies are supported but black-box reuse and hierarchical structuring are not.

The Standard ML language [36] has a very sophisticated module system which supports functions from modules to modules. There is again a namespace of modules, but modules can be nested hierarchically. Connections can be specified by components of module X referring to a previously defined module Y by name. Connections can also be specified by defining a *functor*, a function from modules to modules: If module X depends on a module named Y, then a functor F can be defined whose meaning is the function (λY.X). The functor F can be applied to other modules to yield new concrete modules. This provides flexibility in linking modules. Although ML supports black-box reuse and hierarchical structuring, mutually recursive modules are not allowed. (Current research is addressing this issue, e.g., [15].)

1.2 Reasonable Goals for a Module Formalism

The wide variety of existing module systems have evolved to satisfy a number of goals. We have designed a formal system, the m-calculus, for specifying and reasoning about the behavior of such module systems. In designing the m-calculus, we believed that it should satisfy as many of the following goals as possible:

Reuse without copying or modification: It should be possible (1) to use an individual module more than once in a program, (2) for each use of a module to be connected to other modules in different ways, and (3) for this to be done without changing or duplicating the source code of the module. This is called "black-box reuse" or *extensibility* [32]. Satisfying this requires that inter-module connections need not be specified inside the modules. We handle this in our m-calculus with *incomplete* (or *abstract*) modules and a *linking* operator.

Modules within the language: It should be possible to represent modules and linking together with the features of a core language in a single formalism. Reasoning about the behavior of real systems requires reasoning about all of the components of the real system simultaneously. Satisfying this goal requires either (1) that the module formalism should be able to represent core language features or (2) that it should be possible to combine the module formalism with formal systems for core languages. For our m-calculus we prefer approach (1) although approach (2) should be possible for many core languages.

First-class modules: It should be possible (1) for linking of modules to depend on arbitrary computations, (2) for modules to be created and loaded dynamically, (3) for modules to be passed as parameters and stored in data structures. This kind of power is necessary for reasoning about dynamic linking, a feature which is used in many C implementations on an *ad hoc* basis and is even appearing in language definitions such as that of Java [25]. Satisfying this requires either that the module formalism should support general computation or that it should be able to interact with the formalism used to represent the core language.

Closer fit to real systems: The module formalism should closely fit the actual features of real systems. For example, this means that the coding of modules and linking via λ-calculus, records, and a fix-point operator is inappropriate and cumbersome for languages with package-based module systems. This also means that the module formalism should have direct support for features of existing module systems such as mutual dependencies between modules as well as hierarchical structuring of modules. Our m-calculus easily models all three styles of module system that were described above. (Note that we do not deal with type issues in this paper.)

Sound and flexible equational reasoning: The module formalism should easily support (1) defining how a particular program will behave and (2) understanding the effect of program transformations. While many techniques have been developed for achieving (1), a particularly simple method is to define a reduction semantics, i.e., to define a set of evaluation contexts and a set of program-to-program rewrite rules. If this method is followed, (2) can be achieved by allowing the use of the rewrite rules in any context, not just in evaluation contexts, provided the consistency of the rules can be established. For our m-calculus, we establish internal consistency of the rewrite rules by proving the system is confluent.

1.3 A More General Notion of Module

The key to achieving the above-mentioned goals in the m-calculus is the use of a more general notion of module together with a linking operation. An incomplete or abstract module (introduced as a *mixin module* or a *mixin* in [4], formalized in a calculus in [5], and related to the notions of mixin in [17, 18, 13, 12]) is a collection of components of which some are exported (externally visible), some are private, and some are declared but not defined. We call the latter *deferred* components. For example, consider the following incomplete modules M_1 and M_2, where `N(f,g,i)` is an expression that depends on `f`, `g`, and `i` and similarly for `O(h)` and `P(f,i)`:

```
M1 = (module                        M2 = (module
        exported f = N(f,g,i)               deferred f
        deferred g                          exported g = P(f,i)
        deferred h                          deferred h
        private i = O(h))                   private i = Q)
```

Although the module components are named, the modules themselves do not bear names, i.e., they are anonymous, like abstractions in the λ-calculus [9]. In the m-calculus, we would write the above as:

$$\begin{aligned} M_1 &= \{\mathrm{f} \triangleright \mathrm{w} = \mathrm{N}(\mathrm{w},\mathrm{x},\mathrm{z}),\ \mathrm{g} \triangleright \mathrm{x} = \bullet,\ \mathrm{h} \triangleright \mathrm{y} = \bullet,\ \_ \triangleright \mathrm{z} = \mathrm{O}(\mathrm{y})\} \\ M_2 &= \{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{P}(\mathrm{w},\mathrm{z}),\ \mathrm{h} \triangleright \mathrm{y} = \bullet,\ \_ \triangleright \mathrm{z} = \mathrm{Q}\} \end{aligned}$$

In the m-calculus, each component has separate external and internal names from different namespaces (like in [27]). The internal names are subject to α-conversion and are necessary to support correctness of substitution in the m-calculus. The private components have only an internal name; the label "_" means "no name". Using standard m-calculus abbreviations, we can write the component $(\_ \triangleright \mathrm{z} = \mathrm{O(y)})$ as simply $(\mathrm{z} = \mathrm{O(y)})$. The component body "$\bullet$" indicates a deferred component where the body needs to be filled in by linking.

The meaning of deferred components is established by the linking operation. The result of the operation of linking M_1 and M_2, written $\mathrm{M}_1 \oplus \mathrm{M}_2$, is the new module M_3:

```
M3 = (module
          exported f = N(f,g,i)
          exported g = P(f,i')
          deferred h
          private i = O(h)
          private i' = Q)
```

In linking, deferred components are *concreted* by exported components of the other module. The two modules must not export components with the same name. Private components get renamed as necessary to avoid conflicts. Mutually recursive intermodule dependencies are supported — the example f and g components above depend on each other. In the m-calculus, M_3 is:

$$\mathrm{M}_3 = \{\mathrm{f} \triangleright \mathrm{w} = \mathrm{N(w,x,z)},\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{P(w,z')},\ \mathrm{h} \triangleright \mathrm{y} = \bullet,\ \_ \triangleright \mathrm{z} = \mathrm{O(y)},\ \_ \triangleright \mathrm{z}' = \mathrm{Q}\}$$

The internal name of a component whose name does not match a component in the other module can be α-converted to a fresh name to avoid conflicts. The example does not illustrate this, but internal names of components with matching external names are α-converted to be the same to enable linking. In the m-calculus, M_3 being the result of $\mathrm{M}_1 \oplus \mathrm{M}_2$ is expressed by the single rewrite step $\mathrm{M}_1 \oplus \mathrm{M}_2 \longrightarrow \mathrm{M}_3$.

In addition to modules (which may be incomplete) and linking, only two other kinds of operations are needed for the m-calculus. One is selecting a component of a module, written $M.f$. The other needed operations are component hiding and sieving, written $M\backslash f$ and $M\backslash{-\mathcal{F}}$, necessary for certain kinds of namespace management. (There is also a "letrec" construct $\langle M \mid D\rangle$ which we could have chosen to encode as $\{f \triangleright x = M, D\}.f$.)

1.4 Contributions of This Paper

In section 2, we define the m-calculus, a calculus with modules and linking as primitive constructs. In the m-calculus modules are *first-class.* In section 3, we illustrate how various program construction mechanisms and module systems can be smoothly encoded in the m-calculus. In section 4, we give an overview of the proof of confluence, the bulk of which is treated in [44]. Confluence shows that equational reasoning via the m-calculus is sensible and well behaved and effectively means that rewriting is "meaning"-preserving. The m-calculus is the first calculus of linking for first-class primitive modules which has been proved confluent. (Modules are not first-class in [14, 35] and rewriting is not proven sound in [5].) In addition, in section 5, we discuss the related work.

As limitations, this paper does not deal with issues of types, strict evaluation, imperative effects, or classes and subclassing. As the λ-calculus serves for functions, the m-calculus serves as a theoretical foundation for examining the essence of modularity and linking. Analyses of further issues can be built on the m-calculus as they have been built on the λ-calculus.

1.5 Acknowledgements

We thank Zena Ariola and Lyn Turbak for inspirational discussions.

2 The m-Calculus

2.1 Syntax: Preterms and Raw Terms

The preterms of the m-calculus (the members of the set PreTerm) are given by the following grammar for M:

$$\begin{array}{rlll}
w, x, y, z & \in & \text{Var} & \text{(variables)} \\
f, g, h & \in & \text{CompName} & \text{(component names)} \\
\mathcal{F} & \subseteq & \text{CompName} & \text{(sets of component names)} \\
F & ::= & f \mid \_ & \text{(component label)} \\
B & ::= & M \mid \bullet & \text{(component body)} \\
c & ::= & (F \triangleright x = B) & \text{(component)} \\
D & ::= & c_1, \ldots, c_n \text{ where } n \geq 0 & \text{(component collection)} \\
M, N & ::= & \;\; x & \text{(variable)} \\
 & & \mid (M \backslash f) & \text{(component hiding)} \\
 & & \mid (M \backslash {-\mathcal{F}}) & \text{(component sieving)} \\
 & & \mid (M \oplus N) & \text{(linking)} \\
 & & \mid (M.f) & \text{(component selection)} \\
 & & \mid \{D\} & \text{(module)} \\
 & & \mid \langle M \mid D \rangle & \text{(letrec)}
\end{array}$$

Let $<$ when used on component names be some strict total order. The following operations on components and component collections are defined. Given a component $c = (F \triangleright x = B)$, we define $\text{Label}(c) = F$, $\text{Name}(c) = \text{Label}(c)$ if $\text{Label}(c) \neq \_$ (otherwise undefined), $\text{Binder}(c) = x$, and $\text{Body}(c) = B$. Given a component collection $D = c_1, \ldots, c_n$, we define $|D| = n$, $D[i] = c_i$ if $1 \leq i \leq n$ and is otherwise undefined, $D[i := c] = c_1, \ldots, c_{i-1}, c, c_{i+1}, \ldots, c_n$ if $1 \leq i \leq n$ (otherwise undefined), $\text{Names}(D) = \{\text{Label}(c_1), \ldots, \text{Label}(c_n)\} \setminus \{\_\}$, and $\text{Binders}(D) = \{\text{Binder}(c_1), \ldots, \text{Binder}(c_n)\}$. Let $D[I] = D[i_1], \ldots, D[i_n]$ where $\{i_1, \ldots, i_n\} = I \cap \{1, \ldots, |D|\}$ and $i_1 < \ldots < i_n$. Let $D[\mathcal{F}] = D[i_1], \ldots, D[i_n]$ where $\{i_1, \ldots, i_n\} = \{\, i \mid \text{Name}(D[i]) \in \mathcal{F} \,\}$ and $\text{Name}(D[i_1]) < \ldots < \text{Name}(D[i_n])$ ("components in D with names in $\mathcal{F}$"). Let $D[-\mathcal{F}] = D[\{\, i \mid \text{Label}(D[i]) = F \notin \mathcal{F} \,\}]$ ("components in D with labels *not* in $\mathcal{F}$").

The following terminology is defined. Let $c = (F \triangleright x = B)$ be a component occurring at the top-level (not nested) in a collection D (i.e., $c = D[i]$ for some

i). If the label $F = \text{Label}(c)$ is a name f (and D belongs to a module), then c can be referred to by its name from outside the module for the purposes of linking, selection, or hiding. In this case we may call f an *external* name to distinguish it from the *binder* x which we may call an *internal* name. If F is the *anonymous marker*, written "_", then c is unnamed and is only accessible (internally) via its binder x. The variable $x = \text{Binder}(c)$ is a *binding occurrence* of x which binds free occurrences of x in the bodies of all of the components of D to the body of c. If D is the environment of a letrec $\langle M \mid D \rangle$, then the binder for x also binds free occurrences of x in M. Non-binding variable occurrences are *normal.* The body $B = \text{Body}(c)$ is either a preterm M or the *empty body*, written "$\bullet$". The component c can be of four possible kinds, one of which will be forbidden:

– If $c = (f \triangleright x = M)$, then c is an *exported* or *output* component.
– If $c = (f \triangleright x = \bullet)$, then c is a *deferred* or *input* component.
– If $c = (\_ \triangleright x = M)$, then c is *private* or a *binding.*
– If $c = (\_ \triangleright x = \bullet)$, then this is an error (forbidden below).

A module with input components is *incomplete* or *abstract* and otherwise is *complete* or *concrete.*

The raw terms of the m-calculus (the members of RawTerm) are the preterms satisfying these conditions: (1) An unnamed component does not have an empty body. (2) Two named components in a collection do not have the same name. (3) Components in a collection bind distinct variables. (4) Components in a letrec environment are bindings (unnamed, non-empty bodies).

We use the following conventions for syntactic abbreviations. When writing a member of Term (cf. Section 2.3), a component $(F \triangleright x = B)$ may be written $(F \triangleright \_ = B)$ if no normal occurrences of x are bound by the component's binder. A component $(\_ \triangleright x = B)$ may be written as $(x = B)$; a component $(f \triangleright \_ = B)$ may be written $(f = B)$. The notation $M \backslash \{f_1, \ldots, f_n\}$ stands for $M \backslash f_1 \backslash f_2 \cdots \backslash f_n$ where $f_1 < \cdots < f_n$. The expression (let $x = M$ in M') stands for $\langle M' \mid x = M \rangle$, provided $x \notin \text{FV}(M)$. Parentheses may be omitted; the possible ambiguities are resolved as by giving ".", "\\", and "\\−" higher precedence than "$\oplus$" and making "$\oplus$" left associative.

The free variables of a raw term are defined thus:

$$\begin{aligned}
\text{FV}(\bullet) &= \emptyset \qquad \text{FV}(x) = \{x\} \\
\text{FV}(M \backslash f) &= \text{FV}(M \backslash{-}\mathcal{F}) = \text{FV}(M.f) = \text{FV}(M) \\
\text{FV}(M_1 \oplus M_2) &= \text{FV}(M_1) \cup \text{FV}(M_2) \\
\text{FV}(\{D\}) &= \text{FV}(D) = (\textstyle\bigcup_{1 \le i \le |D|} \text{FV}(\text{Body}(D[i]))) \setminus \text{Binders}(D) \\
\text{FV}(\langle M \mid D \rangle) &= (\text{FV}(M) \setminus \text{Binders}(D)) \cup \text{FV}(D)
\end{aligned}$$

The expression $\text{Capture}_x(M)$ denotes the set of bound variables in raw term M whose binding scope includes a free occurrence of the specific variable x. The operation $M[\![x := y]\!]$ renames to y all free occurrences of the variable x in M that are not in the scope of a binding of y.

A distinguished variable $\Box$, which is forbidden from being bound, is used as the *context hole.* A *context* is a raw term with one occurrence of $\Box$. Let C be a metavariable over contexts. The result of replacing the hole in C by the raw term M (*without* any variable renaming) is written $C[M]$.

2.2 Semantics: Structural and Computational Rewriting on Raw Terms

A rule "$X \rightsquigarrow Y$ if Z" is a schema which defines a *contraction* relation $\rightsquigarrow$ such that $M \rightsquigarrow N$ iff replacing the metavariables in X, Y, and Z by syntactic constructs of the appropriate sort yields, respectively, the terms M and N and a true proposition. A rule schema of the form $D \rightsquigarrow D'$ abbreviates the pair of rule schemas $\{D\} \rightsquigarrow \{D'\}$ and $\langle M \mid D\rangle \rightsquigarrow \langle M \mid D'\rangle$. If a rewrite relation $\longrightarrow$ is the *contextual closure* of a contraction relation $\rightsquigarrow$, this means that $\longrightarrow$ is the least relation such that $M \rightsquigarrow N$ implies $C[M] \longrightarrow C[N]$ for any context C.

The structural rewrite rules will use the following auxiliary definitions:

$$\mathrm{UnsafeNames}(x, D) = \textstyle\bigcup_{1\le i\le |D|} \mathrm{Capture}_x(\mathrm{Body}(D[i])) \cup \mathrm{FV}(D) \cup \mathrm{Binders}(D)$$

$$\mathrm{UnsafeNames}(x, \{D\}) = \mathrm{UnsafeNames}(x, D)$$

$$\mathrm{UnsafeNames}(x, \langle M \mid D\rangle) = \mathrm{Capture}_x(M) \cup \mathrm{FV}(M) \cup \mathrm{UnsafeNames}(x, D)$$

$$\mathrm{BinderRenamed}(i, x, y, D, D') \iff \left(\begin{array}{l} \quad\;\; D \;= (F_1 \rhd x_1 = B_1), \ldots ,(F_i \rhd x = B_i), \ldots ,(F_n \rhd x_n = B_n), \\ \text{and } D' = (F_1 \rhd x_1 = B'_1), \ldots ,(F_i \rhd y = B'_i), \ldots ,(F_n \rhd x_n = B'_n), \\ \text{and } B'_j = B_j[\![x := y]\!] \text{ for } 1 \le j \le n \end{array}\right)$$

The structural rewrite rules are as follows:

$$(\alpha\text{-}\mathbf{letrec}) \qquad \langle M \mid D\rangle \rightsquigarrow \langle M[\![x := y]\!] \mid D'\rangle \quad \text{if } \begin{cases} y \notin \mathrm{UnsafeNames}(x, \langle M \mid D\rangle), \\ \mathrm{BinderRenamed}(i, x, y, D, D') \end{cases}$$

$$(\alpha\text{-}\mathbf{module}) \qquad \{D\} \rightsquigarrow \{D'\} \quad \text{if } \begin{cases} y \notin \mathrm{UnsafeNames}(x, \{D\}), \\ \mathrm{BinderRenamed}(i, x, y, D, D') \end{cases}$$

$$(\mathbf{comp\text{-}order}) \qquad D_1, c_1, D_2, c_2, D_3 \rightsquigarrow D_1, c_2, D_2, c_1, D_3$$

$$(\mathbf{link\text{-}commute}) \; M_1 \oplus M_2 \rightsquigarrow M_2 \oplus M_1$$

The computational rewrite rules, which are presented in Figure 1, use the following auxiliary definitions. The expression PickBody(B, B') yields B if $B' = \bullet$, B' if $B = \bullet$, and is otherwise undefined. DependsOn$_D$ is the least transitive, reflexive relation on $\{1, \ldots , |D|\}$ such that for all $i, j \in \{1, \ldots , |D|\}$,

$$\mathrm{DependsOn}_D(i, j) \Longleftarrow \left(\begin{array}{c} \mathrm{Binder}(D[j]) \in \mathrm{FV}(\mathrm{Body}(D[i])) \\ \text{or } (\mathrm{Body}(D[i]) = \bullet \text{ and } \mathrm{Label}(D[j]) \neq \_) \end{array}\right)$$

The structural and computational contraction relations, $\rightsquigarrow_{\mathrm{s}}$ and $\rightsquigarrow_{\mathrm{c}}$, are respectively the unions of the contraction relations of the structural and computational rules. The structural and computational rewrite relations, $\dashrightarrow_{\mathrm{s}}$ and

$$(\textbf{link}) \quad \{D\} \oplus \{D'\} \rightsquigarrow \{D[-\mathcal{F}], D'[-\mathcal{F}], D''\}$$
$$\text{if } \begin{cases} \mathcal{F} = \{f_1, \ldots, f_n\} = \text{Names}(D) \cap \text{Names}(D'), \\ \text{Binders}(D[-\mathcal{F}]) \cap (\text{Binders}(D') \cup \text{FV}(D')) = \emptyset, \\ \text{Binders}(D'[-\mathcal{F}]) \cap (\text{Binders}(D) \cup \text{FV}(D)) = \emptyset, \\ D[\mathcal{F}] \ = (f_1 \triangleright x_1 = B_1), \ldots, (f_n \triangleright x_n = B_n), \\ D'[\mathcal{F}] = (f_1 \triangleright x_1 = B'_1), \ldots, (f_n \triangleright x_n = B'_n), \\ D'' \quad = (f_1 \triangleright x_1 = B''_1), \ldots, (f_n \triangleright x_n = B''_n), \\ B''_i = \text{PickBody}(B_i, B'_i) \text{ is defined for } 1 \leq i \leq n \end{cases}$$

$$(\textbf{subst}) \quad D \rightsquigarrow D[i := (F_i \triangleright x_i = C[M_j])]$$
$$\text{if } \begin{cases} D[i] = (F_i \triangleright x_i = C[x_j]), \\ D[j] = (F_j \triangleright x_j = M_j), \\ \text{Capture}_\square(C) \cap (\{x_j\} \cup \text{FV}(M_j)) = \emptyset, \\ \text{not DependsOn}_D(j, i) \end{cases}$$

$$(\textbf{subst-letrec}) \quad \langle C[x] \mid D \rangle \rightsquigarrow \langle C[M] \mid D \rangle$$
$$\text{if } \begin{cases} D[i] = ({\_} \triangleright x = M) \text{ for some } i, \\ \text{Capture}_\square(C) \cap (\{x\} \cup \text{FV}(M)) = \emptyset \end{cases}$$

$$(\textbf{select}) \quad \{D\}.f \rightsquigarrow \langle x_i \mid D' \rangle$$
$$\text{if } \begin{cases} D \ = (F_1 \triangleright x_1 = M_1), \ldots, (f \triangleright x_i = M_i), \ldots, (F_n \triangleright x_n = M_n), \\ D' = ({\_} \triangleright x_1 = M_1), \ldots, \ ({\_} \triangleright x_i = M_i), \ldots, ({\_} \triangleright x_n = M_n) \end{cases}$$

$$(\textbf{gc-module}) \quad \{D\} \rightsquigarrow \{D[I]\}$$
$$\text{if } \begin{cases} I, J \text{ partition } \{1, \ldots, |D|\}, \\ J \neq \emptyset, \\ \text{Binders}(D[J]) \cap \text{FV}(D[I]) = \emptyset, \\ \text{Names}(D[J]) = \emptyset \end{cases}$$

$$(\textbf{gc-letrec}) \quad \langle M \mid D \rangle \rightsquigarrow \langle M \mid D[I] \rangle$$
$$\text{if } \begin{cases} I, J \text{ partition } \{1, \ldots, |D|\}, \\ J \neq \emptyset, \\ \text{Binders}(D[J]) \cap (\text{FV}(M) \cup \text{FV}(D[I])) = \emptyset \end{cases}$$

$$(\textbf{empty-letrec}) \quad \langle M \mid \rangle \rightsquigarrow M$$

$$(\textbf{closure}) \quad \langle \{D\} \mid D' \rangle \rightsquigarrow \{D, D'\}$$
$$\text{if } \begin{cases} |D'| > 0, \\ \text{Binders}(D) \cap (\text{Binders}(D') \cup \text{FV}(D')) = \emptyset \end{cases}$$

$$(\textbf{hide-present}) \quad \{D[i := (f \triangleright x = M)]\} \backslash f \rightsquigarrow \{D[i := ({\_} \triangleright x = M)]\}$$

$$(\textbf{hide-absent}) \quad \{D\} \backslash f \rightsquigarrow \{D\}$$
$$\text{if } f \notin \text{Names}(D)$$

$$(\textbf{sieve}) \quad \{D\} \backslash {-\mathcal{F}} \rightsquigarrow \{D'\}$$
$$\text{if } \begin{cases} D \ = (F_1 \triangleright x_1 = B_1), \ldots, (F_n \triangleright x_n = B_n), \\ D' = (F'_1 \triangleright x_1 = B_1), \ldots, (F'_n \triangleright x_n = B_n) \\ F'_i = \begin{cases} {\_} & \text{if } F_i \notin \mathcal{F} \text{ and } B_i \neq \bullet \\ F_i & \text{if } F_i \in \mathcal{F} \end{cases} \qquad \text{for } 1 \leq i \leq n \end{cases}$$

Fig. 1. The computational rewrite rules.

$\dashrightarrow_{\mathrm{c}}$, are the contextual closures of $\rightsquigarrow_{\mathrm{s}}$ and $\rightsquigarrow_{\mathrm{c}}$, respectively. The structural equivalence relation, $=_{\mathrm{s}}$, is the transitive, reflexive, and symmetric closure of $\dashrightarrow_{\mathrm{s}}$. The (combined) contraction relation on raw terms is $\rightsquigarrow \; = \; \rightsquigarrow_{\mathrm{s}} \cup \rightsquigarrow_{\mathrm{c}}$ and the (combined) rewrite relation on raw terms is $\dashrightarrow \; = \; \dashrightarrow_{\mathrm{s}} \cup \dashrightarrow_{\mathrm{c}}$. The relations $\dashrightarrow\!\!\!\!\twoheadrightarrow_{\mathrm{s}}$, $\dashrightarrow\!\!\!\!\twoheadrightarrow_{\mathrm{c}}$, and $\dashrightarrow\!\!\!\!\twoheadrightarrow$ are the transitive, reflexive closures respectively of $\dashrightarrow_{\mathrm{s}}$, $\dashrightarrow_{\mathrm{c}}$, and $\dashrightarrow$.

While variables are subject to α-conversion, component names are not. This is similar to the way that a linker freely relocates (rename) offsets (internal names) within object files as necessary but does not generally rename symbol table entries (external names).

In the presence of cyclic bindings, the usual meta-level substitution and explicit substitution both result in size explosions and generally fail to provide the desired equations between programs. To avoid these difficulties, unlike the calculus of Ancona and Zucca [5], the m-calculus substitutes for one target at a time (via the (**subst**) and (**subst-letrec**) rules) in a style pioneered by Ariola, Blom, and Klop [8, 6, 7]. The m-calculus letrec contruct is, in a sense, a delayed substitution that allows avoiding duplication when a component is selected from a module.

The (**subst**) rule in Figure 1 uses the notion of one component of a collection *depending* on another to exclude certain rewriting possibilities. Without this condition of the (**subst**) rule, the m-calculus would not be confluent and would need a more complicated method as in [35] to prove soundness. Read $\mathrm{DependsOn}_D(j, i)$ as "component $D[j]$ depends on component $D[i]$ in collection D". The first condition of $\mathrm{DependsOn}_D$ handles syntactically evident dependencies. The second condition handles the possibility that a dependency will arise after linking the module $\{D\}$ with another module. Every input component is presumed to (potentially) depend on every output component, because there is always a module to link with that will cause the dependency to become real.

Most of the side conditions of the computational rules which concern the names of bound variables can be met by applying the structural rules first. This is the case for the use of Binders by (**link**) and (**closure**), the use of Capture by (**subst**) and (**subst-letrec**), and the way that (**link**) ensures that the binders of common components have the same name before linking. The side condition in (**closure**) that the component collection is non-empty merely avoids a trivial critical pair with (**empty-letrec**), making proofs easier.

The possible dynamic *errors* that can occur during computation in the m-calculus are (1) linking two modules whose output components are not disjoint, (2) selecting a component from an incomplete module, (3) selecting a component named f from a module which has no component named f, (4) hiding an input component, and (5) sieving out an input component. The following are examples of each of the kinds of errors:

$$
\begin{array}{ll}
(1) & \{\mathrm{f} \rhd \mathrm{w} = \bullet,\ \mathrm{g} \rhd \mathrm{x} = \mathrm{M}\} \oplus \{\mathrm{f} \rhd \mathrm{y} = \mathrm{N},\ \mathrm{g} \rhd \mathrm{z} = \mathrm{N}'\} \\
(2) & \{\mathrm{f} \rhd \mathrm{w} = \bullet,\ \mathrm{g} \rhd \mathrm{x} = \mathrm{M}\}.\mathrm{g} \\
(3) & \{\mathrm{f} \rhd \mathrm{w} = \mathrm{M},\ \mathrm{g} \rhd \mathrm{x} = \mathrm{N}\}.\mathrm{h} \\
(4) & \{\mathrm{f} \rhd \mathrm{w} = \bullet,\ \mathrm{g} \rhd \mathrm{x} = \mathrm{N}\}\backslash \mathrm{f} \\
(5) & \{\mathrm{f} \rhd \mathrm{w} = \bullet,\ \mathrm{g} \rhd \mathrm{x} = \mathrm{N}\}\backslash{-}\{\mathrm{g}\}
\end{array}
$$

2.3 The Calculus: Terms and Rewriting

The actual m-calculus is defined as $\mathcal{M} = (\text{Term}, \longrightarrow) = (\text{RawTerm}, \dashrightarrow_{\text{c}})/ =_{\text{s}}$. By this we mean that:

- The set Term of (real) terms is the set of equivalence classes of the raw terms under $=_{\text{s}}$ (the structural equivalence relation).
- A term $[M]_{=_{\text{s}}}$ (the equivalence class of raw term M under $=_{\text{s}}$) rewrites to a term $[N]_{=_{\text{s}}}$, written $[M]_{=_{\text{s}}} \longrightarrow [N]_{=_{\text{s}}}$, iff there are raw terms $M' \in [M]_{=_{\text{s}}}$ and $N' \in [N]_{=_{\text{s}}}$ such that $M' \dashrightarrow_{\text{c}} N'$.

We assume throughout that raw terms are implicitly coerced to (real) terms when placed in a context requiring a term, e.g., $M \longrightarrow N$ means $[M]_{=_{\text{s}}} \longrightarrow [N]_{=_{\text{s}}}$. Let $\twoheadrightarrow$ be the transitive, reflexive closure of $\longrightarrow$.

3 Encoding Features in the m-Calculus

This section illustrates smooth encodings of various program construction mechanisms in the m-calculus.

3.1 Functions (λ-Calculus)

We define λ-calculus as syntactic sugar for m-calculus terms as follows, where "arg" and "res" are fixed component names (meaning "argument" and "result"):

$$\begin{aligned}(\lambda x.M) &= \{\text{arg} \triangleright x = \bullet,\ \text{res} = M\}\\ (MM') &= (M \oplus \{\text{arg} = M'\}).\text{res}\end{aligned}$$

This encoding is faithful to the meaning of the λ-calculus. We can verify the simulation of β-reduction as follows (where $M[x := M']$ is defined appropriately):

$$\begin{array}{lll}
 & & (\lambda x.M)M' \\
 & = & (\{\text{arg} \triangleright x = \bullet,\ \text{res} = M\} \oplus \{\text{arg} = M'\}).\text{res} \\
 & = & (\{\text{arg} \triangleright x = \bullet,\ \text{res} \triangleright y = M\} \oplus \{\text{arg} \triangleright x = M'\}).\text{res} \\
 & & \quad \text{where } y \notin \text{FV}(M) \cup \text{FV}(M') \text{ and } x \notin \text{FV}(M') \\
(\textbf{link}) & \longrightarrow & \{\text{arg} \triangleright x = M',\ \text{res} \triangleright y = M\}.\text{res} \\
(\textbf{select}) & \longrightarrow & \langle y \mid x = M', y = M\rangle \\
(\textbf{subst-letrec}) & \longrightarrow & \langle M \mid x = M', y = M\rangle \\
(\textbf{gc-letrec}) & \longrightarrow & \langle M \mid x = M'\rangle \\
(\textbf{subst-letrec}) & \twoheadrightarrow & \langle M[x := M'] \mid x = M'\rangle \\
(\textbf{gc-letrec}) & \longrightarrow & \langle M[x := M'] \mid \rangle \\
(\textbf{empty-letrec}) & \longrightarrow & M[x := M']
\end{array}$$

This encoding is similar to an independently developed encoding in [5]. It is only superficially related to the encoding of λ-calculus in ς-calculus [3].

3.2 Records and Record Operations

By the syntactic abbreviations defined in Section 2, record syntax is already accepted by the m-calculus. Furthermore, the expected rewrite rule for selection is simulated.

$$\{f_1 = M_1, \ldots, f_n = M_n\}.f_i \twoheadrightarrow M_i \quad \text{if } 1 \le i \le n$$

The simulation uses (**select**), (**gc-letrec**) (which can be applied because the internal names are not used), and (**empty-letrec**).

3.3 Objects (ς-Calculus)

The following record-of-methods encoding for the ς-calculus [3] works fine. We write "!" for the method invocation operator to avoid confusion with our component selection operator ".".

$$\begin{aligned} [f_1 = \varsigma(x)M_1, \ldots, f_n = \varsigma(x)M_n] &= \{f_1 = \lambda x.M_1, \ldots, f_n = \lambda x.M_n\} \\ (M \Leftarrow f = \varsigma(x)M') &= M\backslash f \oplus [f = \varsigma(x)M'] \\ M!f &= (\text{let } x = M \text{ in } (x.f)x) \quad \text{where } x \text{ is fresh} \end{aligned}$$

It is not hard to verify that the rewrite rules of the ς-calculus are simulated:

$$\begin{aligned} &M!f_i \twoheadrightarrow M_i[x := M] \\ &\quad \text{where } M = [f_1 = \varsigma(x)M_1, \ldots, f_n = \varsigma(x)M_n] \text{ and } 1 \le i \le n \\ &[f_1 = \varsigma(x)M_1, \ldots, f_n = \varsigma(x)M_n] \Leftarrow f_i = \varsigma(x)M' \\ &\quad \twoheadrightarrow [f_1 = \varsigma(x)M_1, \ldots, f_i = \varsigma(x)M', \ldots, f_n = \varsigma(x)M_n] \quad \text{where } 1 \le i \le n \end{aligned}$$

Of course, the real difficulty in dealing with objects is not in expressing their computational meaning but rather in devising the type system, an issue which we do not address in this paper.

3.4 Modules

C-style The m-calculus directly supports the modules of C-like languages. (The call-by-value evaluation and imperative features of C are left to future work.) Each object file O can be represented as a module M, and the linking of the modules $M_1, \ldots, M_n$ to form a program is represented as $P = (M_1 \oplus \ldots \oplus M_n)$. Invoking the program start routine is represented as (P.main).

Package-style For the package style of module system, a module named A which imports modules named $B_1, \ldots, B_n$ and exports entities named $f_1, \ldots, f_m$ is represented by an m-calculus module with one output component named A, and n input components named $B_1, \ldots, B_n$. The output component is in turn a module with n output components named $f_1, \ldots, f_m$ and some number of private components. The linking of modules $M_1, \ldots, M_n$ to form a program is again represented as $P = (M_1 \oplus \ldots \oplus M_n)$. Invoking the start routine of the program is now represented as (P.Main.main), i.e., there is a distinguished module named "Main" which must export a component named "main".

Consider for example the following Haskell program, where `P(A.f)` is an expression mentioning `A.f` and similarly for `Q(B.f,B.g)` and `N`:

```
module A (f) where
 f = N

module B (f, g) where
 import A
 g = P(A.f)
```

```
module Main (main) where
 import qualified B
 f = 5
 main = Q(B.f,B.g,f)
```

This program can be encoded in the m-calculus with these three modules, where A, B, main, and Main are component names:

$$\begin{aligned}
\mathrm{M_A} &= \{\mathrm{A} = \{\mathrm{f} = \mathrm{N}\}\}\\
\mathrm{M_B} &= \{\mathrm{A} \triangleright \mathrm{x} = \bullet,\ \mathrm{B} = \{\mathrm{g} = \mathrm{P}(\mathrm{x.f}),\ \mathrm{f} = \mathrm{x.f}\}\}\\
\mathrm{M_{Main}} &= \{\mathrm{B} \triangleright \mathrm{x} = \bullet,\ \mathrm{Main} = \{\mathrm{y} = 5,\ \mathrm{main} = \mathrm{Q}(\mathrm{x.f}, \mathrm{x.g}, \mathrm{y})\}\}
\end{aligned}$$

Note that the unexported "f" definition in Main is handled by a private component, so a variable "y" must be used instead of a component name. We can check the meaning of the program by rewriting:

$$\begin{aligned}
&(\mathrm{M_A} \oplus \mathrm{M_B} \oplus \mathrm{M_{Main}})\\
\twoheadrightarrow\ &\begin{Bmatrix}\mathrm{A} \triangleright \mathrm{x} = \{\mathrm{f} = \mathrm{N}\},\ \mathrm{B} \triangleright \mathrm{z} = \{\mathrm{g} = \mathrm{P}(\mathrm{x.f}),\ \mathrm{f} = \mathrm{x.f}\}, \\ \mathrm{Main} = \{\mathrm{y} = 5,\ \mathrm{main} = \mathrm{Q}(\mathrm{z.f}, \mathrm{z.g}, \mathrm{y})\}\end{Bmatrix}\\
\twoheadrightarrow\ &\{\mathrm{A} = \{\mathrm{f} = \mathrm{N}\},\ \mathrm{B} = \{\mathrm{g} = \mathrm{P}(\mathrm{N}),\ \mathrm{f} = \mathrm{N}\},\ \mathrm{Main} = \{\mathrm{main} = \mathrm{Q}(\mathrm{N}, \mathrm{P}(\mathrm{N}), 5)\}\}
\end{aligned}$$

Thus, the overall meaning of the program is given by:

$$(\mathrm{M_A} \oplus \mathrm{M_B} \oplus \mathrm{M_{Main}}).\mathrm{Main.main} \twoheadrightarrow \mathrm{Q}(\mathrm{N}, \mathrm{P}(\mathrm{N}), 5)$$

In the Haskell example above, we used qualified names of the form `A.f`. In module `B` we could have used the unqualified name `f` to refer to the entity `A.f`. When a module imports more than one other module, a Haskell implementation uses its knowledge of the imported modules to determine the correct meaning of unqualified names. To encode Haskell modules into the m-calculus, we could use a translation that fully qualifies all names in each using information about the entire program.

However, it is desirable to reason about unqualified names in order to reason about modules separately. Consider for example the above Haskell program with module `B` replaced by the following modules:

```
module B (f, g, i) where
 import A
 import C
 i = 10
 g = P(f,h,i)
```

```
module C (h) where
 h = R
```

The name `f` in module `B` will end up referring to `A.f`, because there is no `C.f`, but this can not be determined without inspecting modules `A` and `C`. The name `i` in module `B` will only be legal if `A.i` and `C.i` do not exist. We can encode these modules as the following (extended) m-calculus modules:

$$\begin{aligned}
\mathrm{M}'_{\mathrm{B}} &= \left\{\begin{matrix}\mathrm{A} \triangleright \mathrm{y} = \bullet,\ \mathrm{C} \triangleright \mathrm{z} = \bullet,\ \mathrm{B} \triangleright \mathrm{w} = \{\mathrm{i} = 10,\ \mathrm{g} = \mathrm{P}(\mathrm{x.f}, \mathrm{x.h}, \mathrm{x.i}),\ \mathrm{f} = \mathrm{x.f}\}, \\ \mathrm{x} = (\mathrm{y} \setminus\!- \{\mathrm{f}, \mathrm{h}, \mathrm{i}\}) \oplus (\mathrm{z} \setminus\!- \{\mathrm{f}, \mathrm{h}, \mathrm{i}\}) \oplus (\mathrm{w} \setminus\!- \{\mathrm{f}, \mathrm{h}, \mathrm{i}\})\end{matrix}\right\} \\
\mathrm{M}_{\mathrm{C}} &= \{\mathrm{C} = \{\mathrm{h} = \mathrm{R}\}\}
\end{aligned}$$

The key idea of this encoding is adding the extra private component defining x to automatically resolve the unqualified names by picking them from whichever module is supplying them. Then we can verify that:

$$(\mathrm{M}_{\mathrm{A}} \oplus \mathrm{M}'_{\mathrm{B}} \oplus \mathrm{M}_{\mathrm{C}} \oplus \mathrm{M}_{\mathrm{Main}}).\mathrm{Main.main} \twoheadrightarrow \mathrm{Q}(\mathrm{N}, \mathrm{P}(\mathrm{N}, \mathrm{R}, 10), 5)$$

In the above example, observe that if M'_{B} is linked with two modules M'_{A} and M'_{C} whose A and C components both supply f, then the linking operation in M'_{B} which yields the private definition of x will get stuck. This corresponds to the fact that this is (usually) illegal in Haskell. (It *is* legal in Haskell for modules `B` and `C` to import module `A` and export `A.f`, and for module `D` to import both `B` and `C` and refer to the unqualified name `f`, because both `B.f` and `C.f` are aliases for `A.f`. It seems that the m-calculus would need to be extended to reason about sharing in order to encode this behavior.)

The Haskell module system has other features such as the ability to list which entities to import from a module, the ability to list entities *not* to import with unqualified names, local aliases for imported modules, and the ability to reexport all of the entities imported from another module. All of these features can be represented in the m-calculus.

ML-style The m-calculus can also represent the type-free aspects of ML-style modules. (The types, call-by-value evaluation, and imperative features of ML are left to future work.) Such module systems provide modules called *structures* as well as a λ-calculus (*functors* and *functor applications*) for manipulating them. A structure is essentially a dependent record; it is dependent in the sense that later fields can refer to the values of earlier fields. A functor is essentially a λ-abstraction whose body denotes a structure; a functor definition is the top-level binding of a functor to its name. ML structures can be encoded in the m-calculus as concrete modules. ML functors and functor applications can be encoded in the m-calculus via the λ-calculus encoding given in Section 3.1.

4 The Well-Behavedness of the Rewrite Rules

This section sketches the proof that the m-calculus is not only *confluent* but that it also satisfies the *finite developments* property. Due to space limitations, the details are only in the long version [44].

Proving these results uses a variation of the m-calculus which adds *redex marks* for tracking residuals of redexes of the computational rules and preventing contraction of freshly created redexes. Redexes of the (**link**), (**select**), (**empty-letrec**), (**closure**), (**hide-present**), (**hide-absent**), and (**sieve**) rules are marked at the root in the usual way. Redexes of (**subst**) and (**subst-letrec**) are marked at the variable which is the substitution target rather than at the root. Redexes of (**gc-module**) and (**gc-letrec**) are also not marked at the root; instead each component that can be garbage collected is marked. All marks are 0 except for substitution marks which must be 1 greater than all of the marks in the substitution source component body. (Due to the side condition on (**subst**) using DependsOn, it is always possible to mark all redexes in a term.)

Strong normalization (termination of rewriting) of the marked m-calculus is proved using a decreasing measure, the multiset of all marks in the term, in the well founded multiset ordering. Weak confluence of the marked m-calculus is proved by several lemmas established by careful case analyses together with a top-level proof structure that separately considers structural and computational rewrite steps. Our proof deals with and accounts for every structural operation (i.e., α-conversion and re-ordering) explicitly.

The combination of strong normalization and weak confluence of the marked m-calculus yields confluence of the marked m-calculus. Then developments are defined as those rewrite sequences of the m-calculus that can be lifted to the marked m-calculus. Using the confluence of the marked m-calculus, we prove that the results of any two coinitial developments can be joined by two further developments. Standard techniques then finish the proof of confluence of the m-calculus. Confluence is shown both for $\longrightarrow$ (on terms) and $\dashrightarrow$ (on raw terms).

5 Related Work

5.1 Calculi with Linking

Cardelli presents a simply-typed linking calculus for outermost-only modules without recursion [14]. Drossopoulou, Eisenbach, and Wragg give a module calculus for reasoning about the quirks of Java [16]. Ancona and Zucca give a calculus for linking modules which, although similar to ours, has a notion of substitution which we believe is less convenient and no published proof of rewriting properties [5]. Earlier, Ancona and Zucca also presented an algebra for simplifying module expressions which is not powerful enough to represent general computation [4]. Machkasova and Turbak give a calculus for linking outermost-only modules in a call-by-value language [35].

From a non-equational-reasoning point of view, Flatt and Felleisen give a calculus of modules with similar capabilities to ours [21]. Glew and Morrisett present a module calculus tailored towards dealing with linking of object files containing assembly-language-level code [24]. Waddell and Dybvig show how to encode modules and linking using Scheme's macro system [42].

5.2 Mixins

Duggan and Sourelis present a system of "mixin modules" which has the unique feature that when both modules have components with the same name, linking the modules results in a form of merging of the same-named components [17, 18]. Bracha and Lindstrom encode mixins using λ-calculus, records, and fix-point operators [13, 12]. Findler and Flatt describe using mixins and incomplete modules in actual programming [19]. Flatt and Krishnamurthi and Felleisen present a calculus with an operational semantics for mixins and classes in the context of Java [22].

5.3 Calculi for Cycles

Inspiring much of our formulation, Ariola and Klop did ground-breaking work on reasoning about λ-terms combined with a construct for mutually recursive definitions [8]. Ariola and Blom refined this work to prove consistency in the absence of confluence [6, 7].

5.4 ML-Style Modules vs. Types

Crary, Harper, and Puri describe how to extend the ML module system to deal with recursion [15]. Earlier work to add first-class modules (i.e., higher-order functors) to ML includes that of Russo [41], Harper and Lillibridge [27, 34], and Leroy [33]. Harper, Mitchell, and Moggi devised the *phase distinction* to show the decidability of type checking for the ML module system [28]. Jones shows how to avoid much of the complexity of typing ML-style modules via higher-order (parametric) signatures [31, 30].

5.5 Types vs. Concatenation and Extension for Records and Objects

When we extend our system with types, we will closely consider previous work on types for record concatenation [43, 29], extensible records [39, 23], and extensible objects [20, 40, 11].

References

[1] Haskell 98: A non-strict, purely functional language. Technical report, The Haskell 98 Committee, 1 Feb. 1999. Currently available at http://haskell.org.
[2] LNCS. Springer-Verlag, 2000.
[3] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
[4] D. Ancona and E. Zucca. An algebra of mixin modules. In F. P. Presicce, editor, *Recent Trends in Algebraic Development Techniques (12th Int'l Workshop, WADT '97 — Selected Papers)*, number 1376 in LNCS, pages 92–106. Springer-Verlag, 1998.

[5] D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Proc. Int'l Conf. on Principles and Practice of Declarative Programming*, LNCS, Paris, France, 29 Sept. – 1 Oct. 1999. Springer-Verlag.

[6] Z. M. Ariola and S. Blom. Cyclic lambda calculi. In *Theoretical Aspects Comput. Softw. : Int'l Conf.*, 1997.

[7] Z. M. Ariola and S. Blom. Lambda calculi plus letrec. Submitted, 3 July 1997.

[8] Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Inf. & Comput.*, 139:154–233, 1997.

[9] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[10] J. G. P. Barnes. *Programming in Ada 95.* Addison-Wesley, 1996.

[11] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping constraints for incomplete objects. *Fundamenta Informaticae*, 199X. To appear.

[12] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity, and Multiple Inheritance.* PhD thesis, Univ. of Utah, Mar. 1992.

[13] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proc. Int'l Conf. Computer Languages*, pages 282–290, 1992.

[14] L. Cardelli. Program fragments, linking, and modularization. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997.

[15] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. ACM SIGPLAN '99 Conf. Prog. Lang. Design & Impl.*, 1997.

[16] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus — towards a model of separate compilation, linking and binary compatibility. In *Proc. 14th Ann. IEEE Symp. Logic in Computer Sci.*, July 1999.

[17] D. Duggan and C. Sourelis. Mixin modules. In *Proc. 1996 Int'l Conf. Functional Programming*, pages 262–273, 1996.

[18] D. Duggan and C. Sourelis. Parameterized modules, recursive modules, and mixin modules. In *ACM SIGPLAN Workshop on ML and its Applications*, 1998.

[19] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. 1998 Int'l Conf. Functional Programming*, 1998.

[20] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[21] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN '98 Conf. Prog. Lang. Design & Impl.*, 1998.

[22] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.

[23] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Univ. of Nottingham, 1996.

[24] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In POPL '99 [38].

[25] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[26] S. P. Harbison. *Modula-3.* Prentice Hall, 1991.

[27] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL '94 [37], pages 123–137.

[28] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Conf. Rec. 17th Ann. ACM Symp. Princ. of Prog. Langs.*, 1990.

[29] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. Technical Report CMU-CS-90-157R, Carnegie Mellon Univ., 2 July 1991.

[30] M. P. Jones. From Hindley-Milner types to first-class structures. In *Proceedings of the Haskell Workshop*, La Jolla, California, U.S.A., 25 June 1995.

[31] M. P. Jones. Using parameterized signatures to express modular structure. In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[32] S. Krishnamurthi and M. Felleisen. Toward a formal theory of extensible software. In *Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Nov. 1998.

[33] X. Leroy. Manifest types, modules, and separate compilation. In POPL '94 [37], pages 109–122.

[34] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, Carnegie Mellon Univ., May 1997.

[35] E. Machkasova and F. Turbak. A calculus for link-time compilation. In *Proc. European Symp. on Programming* [2].

[36] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1990.

[37] *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, 1994.

[38] *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, 1999.

[39] D. Rémy. Projective ML. In *Proc. 1992 ACM Conf. LISP Funct. Program.*, 1992.

[40] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 199X. To appear.

[41] C. V. Russo. *Types for Modules.* PhD thesis, Univ. of Edinburgh, 1998.

[42] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In POPL '99 [38].

[43] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. 4th Ann. Symp. Logic in Computer Sci.*, pages 92–97, Pacific Grove, CA, U.S.A., June 5–8 1989. IEEE Comput. Soc. Press.

[44] J. B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules (long version). A short version is [45]. Full paper with three appendices for proofs, Aug. 1999.

[45] J. B. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Proc. European Symp. on Programming* [2]. A long version is [44].