Confluent Equational Reasoning for Linking with First-Class Primitive Modules

Joe Wells*
Heriot-Watt University
http://www.cee.hw.ac.uk/~jbw/

René Vestergaard
Heriot-Watt University
http://www.cee.hw.ac.uk/~jrvest/

August 25, 1999

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 present the m-calculus, a calculus where the primitive constructs are modules, linking, and the selection and hiding of module components. In addition to supporting equational reasoning about modules and linking, the m-calculus allows smooth encodings of software structuring tools such as the λ-calculus, mutually recursive definitions, records (including operations like extension and concatenation), and objects. The m-calculus is extremely well behaved — we show not only that the m-calculus is *confluent* but also that it satisfies the *strong finite developments* property.

1 Introduction

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 [Bar96], Modula-3 [Har91], and Haskell [Jon99] 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 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 [MTHM90] 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 [CHP99]).

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:

*Supported by EPSRC grant GR/L 36963 and NSF grants CCR–9417382 and CCR–9806747.

- **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 the source code of the module or duplicating the module. This is called "black-box reuse" or *extensibility* [KF98]. 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 choose approach (1) although it should be possible to use approach (2) for many core language formalisms.

- **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 [GJS96]. 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 and records 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 typing issues in this paper.)

- **Sound and flexible equational reasoning:** The module formalism should provide simple mechanisms for (1) defining how a particular program will behave and (2) understanding the effect of program transformations. Many techniques have been developed for achieving (1), but 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 (called a *mixin module* or a *mixin* in [AZ98] and closely related to the notions of mixin in [DS96, DS98, BL92, Bra92]) 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. Consider the following example 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(i)`:

M_1 =
```
(module
   exported f = N(f,g,i)
   deferred g
   deferred h
   private i = O(h)
 end)
```

M_2 =
```
(module
   deferred f
   exported g = P(f,i)
   deferred h
   private i = Q
 end)
```

Notice that although the module components are named, the modules themselves do not bear names, i.e., they are anonymous, like abstractions in the λ-calculus [Bar84]. In the m-calculus, we would write the above as:

$$\mathrm{M}_1 = \{\mathrm{f} \triangleright \mathrm{w} = \mathrm{N}(\mathrm{w},\mathrm{x},\mathrm{z}),\ \mathrm{g} \triangleright \mathrm{x} = \bullet,\ \mathrm{h} \triangleright \mathrm{y} = \bullet,\ \_ \triangleright \mathrm{z} = \mathrm{O}(\mathrm{y})\}$$
$$\mathrm{M}_2 = \{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{P}(\mathrm{w},\mathrm{z}),\ \mathrm{h} \triangleright \mathrm{y} = \bullet,\ \_ \triangleright \mathrm{z} = \mathrm{Q}\}$$

Notice that in the m-calculus, each component has separate external and internal names (like in [HL94]). 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 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
      end)
```

Notice that in linking, deferred components are *concreted* by exported components of the other module. We require that the two modules do not export components with the same name. Notice that private components get renamed as necessary so as not to cause conflicts. Notice also that mutually recursive intermodule dependencies are supported: The `f` and `g` components depend on each other in the linking result. We would write M_3 in the m-calculus as:

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

Notice that the internal name of a component whose name does not match a component in the other module is α-converted to a fresh name. 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 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 operations are needed for the m-calculus. One of these operations is selecting a component of a module, written $M.f$. The other needed operation is component hiding, written $M \backslash f$, necessary for encoding certain kinds of namespace management. Our expression of the meaning of $M.f$ (given by rewrite rules) uses a recursive binding construct, written $\langle M \mid D \rangle$, where D is a set of bindings. However, we use this "letrec" construct only for convenience in our presentation and proofs, as we could have chosen to write $\langle M \mid D \rangle$ as $\{f \triangleright x = M, D\}.f$ for fresh x and arbitrarily chosen f.

1.4 Contributions of This Paper

The significance of this paper is as follows:

- In section 2, we define the m-calculus, a calculus with modules and linking as primitive constructs. The m-calculus gives a theoretical foundation in the style of the λ-calculus for reasoning about modules and linking. Furthermore, it supports *first-class* modules.
- In section 3, we illustrate how various program construction mechanisms can be smoothly encoded in the m-calculus.
- In section 4, we prove the m-calculus to be confluent, thereby establishing that it is internally consistent. The m-calculus is the first calculus of linking for first-class primitive modules which has been proved confluent.

In addition, in section 5, we discuss the related work.

1.5 Acknowledgements

The first author would like to thank Zena Ariola for explaining the proper way to handle sets of mutually recursive definitions during her visit to the Church Project in Boston in Spring 1996. He would also like to thank Lyn Turbak for mentioning an idea which helped trigger much of this work.

2 The m-Calculus

2.1 Syntax: Preterms and Raw Terms

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

Figure 1: The preterms of the m-calculus (the set PreTerm).

The preterms of the m-calculus (the members of the set PreTerm) are given by the grammar for M in Figure 1. Various operations on individual components and component collections are defined in Figure 2. Let $<$ when used on component names (the members of CompName) be some strict total order. Figure 3 defines standard syntactic abbreviations for the m-calculus.

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 called *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 $c = (f \triangleright x = M)$ (named, body is preterm), then we call c an *exported* or *output* component.
- If $c = (f \triangleright x = \bullet)$ (named, body is empty), then we call c a *deferred* or *input* component.
- If $c = (\_ \triangleright x = M)$ (unnamed, body is preterm), then we call c a *private* component or a *binding*.
- If $c = (\_ \triangleright x = \bullet)$ (unnamed, body is empty), then this is an error. It will be forbidden below.

If a module has input components, it is *incomplete* or *abstract* and otherwise it is *complete* or *concrete*.

The raw terms of the m-calculus (the members of RawTerm) are the preterms which satisfy certain well-formedness conditions. These conditions are inconvenient to specify in a context-free grammar, so we give them as follows:

- An unnamed component does not have an empty body. Formally, in any c, it holds that either $\text{Label}(c) \neq \_$ or $\text{Body}(c) \neq \bullet$.
- Two named components in a collection do not have the same name. Formally, in any D, it holds that $\text{Label}(D[i]) \neq \text{Label}(D[j])$ if $\text{Label}(D[i]) \neq \_$ and $1 \leq i < j \leq |D|$.
- Components in a collection bind distinct variables. Formally, in any D, it holds that $\text{Binder}(D[i]) \neq \text{Binder}(D[j])$ if $1 \leq i < j \leq |D|$.
- Components in a letrec environment are bindings (unnamed, non-empty bodies). Formally, for any subterm $\langle M \mid D \rangle$, it holds that $\text{Label}(D[i]) \neq \_$ and $\text{Body}(D[i]) \neq \bullet$ for $1 \leq i \leq |D|$.

- Given a component $c = (F \triangleright x = B)$, we define:
  - $\text{Label}(c) = F$,
  - $\text{Name}(c) = \begin{cases} \text{Label}(c) & \text{if Label}(c) \neq \_, \\ \text{undefined} & \text{otherwise,} \end{cases}$
  - $\text{Binder}(c) = x$,
  - $\text{Body}(c) = B$.
- Given a component collection $D = c_1, \ldots, c_n$, we define:
  - $|D| = n$,
  - $D[i] = \begin{cases} c_i & \text{if } 1 \leq i \leq n, \\ \text{undefined} & \text{otherwise,} \end{cases}$
  - $D[i := c] = \begin{cases} c_1, \ldots, c_{i-1}, c, c_{i+1}, \ldots, c_n & \text{if } 1 \leq i \leq n, \\ \text{undefined} & \text{otherwise,} \end{cases}$
  - $\text{Labels}(D) = \{\text{Label}(c_1), \ldots, \text{Label}(c_n)\}$,
  - $\text{Names}(D) = \text{Labels}(D) \setminus \{\_\}$,
  - $\text{Binders}(D) = \{\text{Binder}(c_1), \ldots, \text{Binder}(c_n)\}$.
- $D[f] = \begin{cases} D[i] & \text{if Label}(D[i]) = f \text{ and Label}(D[j]) \neq f \text{ for } 1 \leq j < i, \\ \text{undefined} & \text{otherwise.} \end{cases}$
- $D[I]$ is the collection $D[i_1], \ldots, D[i_n]$ where $\{i_1, \ldots, i_n\} = I \cap \{1, \ldots, |D|\}$ and $i_1 < \ldots < i_n$.
- $D[\mathcal{F}]$ is the collection $D[i_1], \ldots, D[i_n]$ 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 $\mathcal{F}$").
- $D[-\mathcal{F}] = D[\{\, i \mid \text{Label}(D[i]) = F \notin \mathcal{F} \,\}]$ ("components in D with labels *not* in $\mathcal{F}$").
- $D, D' = \begin{cases} D & \text{if } |D'| = 0 \quad \text{(i.e., the comma disappears),} \\ D' & \text{if } |D| = 0 \quad \text{(i.e., the comma disappears),} \\ D, D' & \text{otherwise.} \end{cases}$

Figure 2: Operations on components and component collections.

- 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.
- The following syntactic abbreviations are defined:

$$\begin{array}{rll} (x = B) & \text{means} & (\_ \triangleright x = B), \\ (f = B) & \text{means} & (f \triangleright \_ = B), \\ M\backslash\{f_1, \ldots, f_n\} & \text{means} & M\backslash f_1 \backslash f_2 \cdots \backslash f_n \quad \text{where } f_1 < \cdots < f_n, \\ (\text{let } x = M \text{ in } M') & \text{means} & \langle M' \mid x = M \rangle \quad \text{if } x \notin \text{FV}(M). \end{array}$$

- Parentheses may be omitted. The possible ambiguities are resolved as follows:

$$\begin{array}{rll} M_1 \oplus M_2.f & \text{means} & M_1 \oplus (M_2.f), \\ M_1 \oplus M_2 \backslash f & \text{means} & M_1 \oplus (M_2 \backslash f), \\ M_1 \oplus M_2 \oplus M_3 & \text{means} & (M_1 \oplus M_2) \oplus M_3. \end{array}$$

Figure 3: Conventions for syntactic abbreviations (syntactic sugar).

$$
\begin{array}{rcl}
\mathrm{FV}(\bullet) & = & \emptyset \\
\mathrm{FV}(x) & = & \{x\} \\
\mathrm{FV}(M\backslash f) & = & \mathrm{FV}(M) \\
\mathrm{FV}(M_1 \oplus M_2) & = & \mathrm{FV}(M_1) \cup \mathrm{FV}(M_2) \\
\mathrm{FV}(M.f) & = & \mathrm{FV}(M) \\
\mathrm{FV}(D) & = & (\bigcup_{1 \leq i \leq |D|} \mathrm{FV}(\mathrm{Body}(D[i]))) \setminus \mathrm{Binders}(D) \\
\mathrm{FV}(\{D\}) & = & \mathrm{FV}(D) \\
\mathrm{FV}(\langle M \mid D \rangle) & = & (\mathrm{FV}(M) \setminus \mathrm{Binders}(D)) \cup \mathrm{FV}(D)
\end{array}
$$

Figure 4: The free variables of a raw term.

$$
\begin{array}{rcl}
\mathrm{Capture}_x(\bullet) & = & \emptyset \\
\mathrm{Capture}_x(y) & = & \emptyset \quad \text{(regardless of whether } x = y\text{)} \\
\mathrm{Capture}_x(M\backslash f) & = & \mathrm{Capture}_x(M) \\
\mathrm{Capture}_x(M_1 \oplus M_2) & = & \mathrm{Capture}_x(M_1) \cup \mathrm{Capture}_x(M_2) \\
\mathrm{Capture}_x(M.f) & = & \mathrm{Capture}_x(M) \\
\mathrm{Capture}_x(D) & = & \begin{cases} \mathrm{Binders}(D) \cup \bigcup_{1 \leq i \leq |D|} \mathrm{Capture}_x(\mathrm{Body}(D[i])) & \text{if } x \in \mathrm{FV}(D), \\ \emptyset & \text{otherwise.} \end{cases} \\
\mathrm{Capture}_x(\{D\}) & = & \mathrm{Capture}_x(D) \\
\mathrm{Capture}_x(\langle M \mid D \rangle) & = & \begin{cases} \mathrm{Binders}(D) \cup \mathrm{Capture}_x(M) \cup \mathrm{Capture}_x(D) & \text{if } x \in \mathrm{FV}(\langle M \mid D \rangle), \\ \emptyset & \text{otherwise.} \end{cases}
\end{array}
$$

Figure 5: Bound variables whose binding scope includes a free occurrence of a specific variable, x.

$$
\begin{array}{rcl}
\bullet[\![x := y]\!] & = & \bullet \\
z[\![x := y]\!] & = & \begin{cases} z & \text{if } x \neq z \\ y & \text{otherwise} \end{cases} \\
(M\backslash f)[\![x := y]\!] & = & M[\![x := y]\!]\backslash f \\
(M_1 \oplus M_2)[\![x := y]\!] & = & (M_1[\![x := y]\!]) \oplus (M_2[\![x := y]\!]) \\
(M.f)[\![x := y]\!] & = & M[\![x := y]\!].f \\
(F \triangleright z = B)[\![x := y]\!] & = & \begin{cases} (F \triangleright z = (B[\![x := y]\!])) & \text{if } z \notin \{x,y\} \\ (F \triangleright z = B) & \text{otherwise} \end{cases} \\
D[\![x := y]\!] & = & \begin{cases} D[1][\![x := y]\!], \ldots, D[|D|][\![x := y]\!] & \text{if } \{x, y\} \cap \mathrm{Binders}(D) = \emptyset, \\ D & \text{otherwise.} \end{cases} \\
\{D\}[\![x := y]\!] & = & \{D[\![x := y]\!]\} \\
\langle M \mid D \rangle[\![x := y]\!] & = & \begin{cases} \langle M[\![x := y]\!] \mid D[\![x := y]\!] \rangle & \text{if } \{x, y\} \cap \mathrm{Binders}(D) = \emptyset, \\ \langle M \mid D \rangle & \text{otherwise.} \end{cases}
\end{array}
$$

Figure 6: Renaming of free variables on raw terms (a restricted form of substitution).

Auxiliary Definitions:

$$\begin{aligned}
\mathrm{UnSafeNewNames}(x, D) &= \textstyle\bigcup_{1 \le i \le |D|} \mathrm{Capture}_x(\mathrm{Body}(D[i])) \cup \mathrm{FV}(D) \cup \mathrm{Binders}(D) \\
\mathrm{UnSafeNewNames}(x, \{D\}) &= \mathrm{UnSafeNewNames}(x, D) \setminus \{x\} \\
\mathrm{UnSafeNewNames}(x, \langle M \mid D \rangle) &= (\mathrm{Capture}_x(M) \cup \mathrm{FV}(M) \cup \mathrm{UnSafeNewNames}(x, D)) \setminus \{x\}
\end{aligned}$$

$$\mathrm{BinderRenamed}(i, x, y, D, D') \iff \begin{pmatrix} D\ = (F_1 \triangleright x_1 = B_1), \ldots, (F_i \triangleright x = B_i), \ldots, (F_n \triangleright x_n = B_n), \\ \text{and } D' = (F_1 \triangleright x_1 = B_1'), \ldots, (F_i \triangleright y = B_i'), \ldots, (F_n \triangleright x_n = B_n'), \\ \text{and } B_j' = B_j[\![x := y]\!] \text{ for } 1 \le j \le n \end{pmatrix}$$

Rewrite Rules:

(α-**letrec**) $\quad \langle M \mid D \rangle \rightsquigarrow \langle M[\![x := y]\!] \mid D' \rangle$

$$\text{where } \begin{cases} y \notin \mathrm{UnSafeNewNames}(x, \langle M \mid D \rangle), \\ \mathrm{BinderRenamed}(i, x, y, D, D') \end{cases}$$

(α-**module**) $\quad \{D\} \rightsquigarrow \{D'\}$

$$\text{where } \begin{cases} y \notin \mathrm{UnSafeNewNames}(x, \{D\}), \\ \mathrm{BinderRenamed}(i, x, y, D, D') \end{cases}$$

(**comp-order**) $\quad D_1, c_1, D_2, c_2, D_3 \rightsquigarrow D_1, c_2, D_2, c_1, D_3$

(**link-commute**) $\quad M_1 \oplus M_2 \rightsquigarrow M_2 \oplus M_1$

Figure 7: The structural rewrite rules.

Figure 4 defines the free variables of a raw term. Figure 5 defines a function $\mathrm{Capture}_x(M)$ which computes the bindings in M which can capture variables in anything which replaces the free variable x. Figure 6 defines an operation $M[\![x := y]\!]$ renaming the free variable x to y in M. This operation is totally defined but only correct when $y \notin \mathrm{Capture}_x(M) \cup \mathrm{FV}(M)$, i.e., when y will not be improperly captured (cf. the last three equations in Figure 6 and also Lemma 3) or confused with another free variable.

We reserve a distinguished variable $\Box$ for use as the *context hole*. We forbid $\Box$ from being bound. A *context* is a raw term with exactly one occurrence of $\Box$. Let C be a meta-variable over contexts. We write $C[M]$ to denote the raw term obtained by syntactically replacing the hole in C by the raw term M (*without* any variable renaming). Note that $\mathrm{Capture}_\Box(C)$ is thus well-defined notation.

2.2 Semantics: Structural and Computational Rewriting on Raw Terms

Several conventions are used in presenting rewrite rules. A rule $X \rightsquigarrow Y$ is a schema which defines a *contraction* relation $\rightsquigarrow$ such that $M \rightsquigarrow N$ iff replacing the metavariables in X and Y by syntactic constructs of the appropriate sort yields the terms M and N, respectively. A rule schema of the form $D \rightsquigarrow D'$ abbreviates the pair of rule schemas $\{D\} \rightsquigarrow \{D'\}$ and $\langle M \mid D \rangle \rightsquigarrow \langle M \mid D' \rangle$. When we say that 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.

Figure 7 presents the structural rewrite rules. The structural contraction relation $\rightsquigarrow_{\mathrm{s}}$ is the union of the contraction relations of all of the structural rules. The structural rewrite relation, $\dashrightarrow_{\mathrm{s}}$, is the contextual closure of $\rightsquigarrow_{\mathrm{s}}$. The structural equivalence relation, $=_{\mathrm{s}}$, is the transitive, reflexive, and symmetric closure of $\dashrightarrow_{\mathrm{s}}$.

We have two comments on the definition of the α-rules in Figure 7. First, observe that 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). Second, the definition of the α-rules depends on the notion of *unsafe* new names for a bound variable x. This notion is necessary to make sure the α-rules have the properties we desire for all of the structural rules: correctness and 1-step reversibility. The names of free variables and nested capturing variable binders are unsafe for reasons of correctness, while binders at the same level are unsafe because of the syntactic restrictions distinguishing raw terms from preterms.

Figure 8 presents the computational rewrite rules. The computational contraction relation $\rightsquigarrow_{\mathrm{c}}$ is the

Auxiliary Definitions:

$$\text{PickBody}(B, B') = \begin{cases} B & \text{if } B' = \bullet, \\ B' & \text{if } B = \bullet, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

DependsOn_D is the least transitive, reflexive relation on $\{1, \ldots, |D|\}$ such that for all $i, j \in \{1, \ldots, |D|\}$,

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

Rewrite Rules:

(**link**) $\{D\} \oplus \{D'\} \rightsquigarrow \{D[-\mathcal{F}], D'[-\mathcal{F}], D''\}$

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

(**subst**) $D \rightsquigarrow D[i := (F_i \triangleright x_i = C[M_j])]$

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

(**subst-letrec**) $\langle C[x] \mid D \rangle \rightsquigarrow \langle C[M] \mid D \rangle$

$$\text{where } \begin{cases} D[i] = (\_ \triangleright x = M) \text{ for some } i, \\ \text{Capture}_{\square}(C) \cap (\{x\} \cup \text{FV}(M)) = \emptyset \end{cases}$$

(**select**) $\{D\}.f \rightsquigarrow \langle x_i \mid D' \rangle$

$$\text{where } \begin{cases} D \ = (F_1 \triangleright x_1 = M_1), \ldots, (f \triangleright x_i = M_i), \ldots, (F_n \triangleright x_n = M_n), \\ D' = (\_ \triangleright x_1 = M_1), \ldots, \ (\_ \triangleright x_i = M_i), \ldots, (\_ \triangleright x_n = M_n) \end{cases}$$

(**gc-module**) $\{D\} \rightsquigarrow \{D[I]\}$

$$\text{where } \begin{cases} I, J \text{ partition } \{1, \ldots, |D|\}, \\ J \neq \emptyset, \\ \text{Binders}(D[J]) \cap \text{FV}(D[I]) = \emptyset, \\ \text{Names}(D[J]) = \emptyset \end{cases}$$

(**gc-letrec**) $\langle M \mid D \rangle \rightsquigarrow \langle M \mid D[I] \rangle$

$$\text{where } \begin{cases} I, J \text{ partition } \{1, \ldots, |D|\}, \\ J \neq \emptyset, \\ \text{Binders}(D[J]) \cap (\text{FV}(M) \cup \text{FV}(D[I])) = \emptyset \end{cases}$$

(**empty-letrec**) $\langle M \mid \rangle \rightsquigarrow M$

(**closure**) $\langle \{D\} \mid D' \rangle \rightsquigarrow \{D, D'\}$

$$\text{where } \begin{cases} |D'| > 0, \\ \text{Binders}(D) \cap (\text{Binders}(D') \cup \text{FV}(D')) = \emptyset \end{cases}$$

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

(**hide-absent**) $\{D\} \backslash f \rightsquigarrow \{D\}$
where $f \notin \text{Labels}(D)$

Figure 8: The computational rewrite rules.

union of the contraction relations of all of the computational rules. The computational rewrite relation $\dashrightarrow_{\mathrm{c}}$ is the contextual closure of $\leadsto_{\mathrm{c}}$.

The (**subst**) rule in Figure 8 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. Read $\mathrm{DependsOn}_D(j, i)$ as "component $D[j]$ depends on component $D[i]$ in collection D". The first condition in the definition of $\mathrm{DependsOn}_D$ handles syntactically evident dependencies. The second condition handles the possibility that there will be a dependency 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.

The possible dynamic errors that can occur during computation can be seen in the definition of the (**link**) and (**select**) rules. The possible *errors* 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, and (4) hiding an input component. Note that the (**link**), (**select**), (**hide-present**), and (**hide-absent**) rules are defined so as to do nothing to errors, i.e., errors are *stuck*. The following are examples of errors:

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

Note that 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 is only there to prevent a trivial critical pair with the (**empty-letrec**) rule. This makes some proofs in Section 4 easier, because the redex/contractum pair uniquely determines which rewrite rule was used.

Let the contraction relation on raw terms be $\leadsto\ =\ \leadsto_{\mathrm{s}} \cup \leadsto_{\mathrm{c}}$ and let the rewrite relation on raw terms be $\dashrightarrow\ =\ \dashrightarrow_{\mathrm{s}} \cup \dashrightarrow_{\mathrm{c}}$. Let $\dashrightarrow\!\!\!\!\!\rightarrow_{\mathrm{s}}$, $\dashrightarrow\!\!\!\!\!\rightarrow_{\mathrm{c}}$, and $\dashrightarrow\!\!\!\!\!\rightarrow$ be the transitive, reflexive closures respectively of $\dashrightarrow_{\mathrm{s}}$, $\dashrightarrow_{\mathrm{c}}$, and $\dashrightarrow$.

Lemma 1 (Closure under Rewriting) *If M is a raw term and $M \dashrightarrow N$, then N is a raw term.*

Proof: By inspection. □

2.3 The Calculus: Terms and Rewriting

Definition 2 (The m-Calculus) *The m-calculus is $\mathcal{M} = (\mathrm{Term}, \longrightarrow) = (\mathrm{RawTerm}, \dashrightarrow_{\mathrm{c}})/=_{\mathrm{s}}$. By this we mean that:*

1. *The set* Term *of (real) terms is the set of equivalence classes of the raw terms under the structural equivalence relation.*

2. *A term M (an equivalence class of raw terms) rewrites to a term N (another equivalence class), written $M \longrightarrow N$, iff there are raw terms $M' \in M$ and $N' \in N$ such that $M' \dashrightarrow_{\mathrm{c}} N'$.*

To write a (real) term M, it suffices to write any raw term $M' \in M$. Whether a raw term M denotes itself or the equivalence class to which it belongs is determined by context. Let $\longrightarrow\!\!\!\!\!\rightarrow$ be the transitive, reflexive closure of $\longrightarrow$.

3 Encoding Features in the m-Calculus

This section illustrates how various program construction mechanisms can be smoothly encoded in the m-calculus.

3.1 Functions (λ-Calculus)

Remember that $(f = M)$ is syntactic sugar for $(f \rhd \_ = M)$ (cf. Figure 3). We can define λ-calculus as syntactic sugar for m-calculus terms as follows, where "arg" and "res" are fixed component names (meaning "argument" and "result"):

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

This encoding is faithful to the meaning of the λ-calculus. It holds that β-reduction is simulated, i.e., the following holds where the substitution $M[x := M']$ is defined appropriately:

$$(\lambda x.M)M' \twoheadrightarrow M[x := M']$$

We can verify the simulation of β-reduction as follows:

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

Furthermore, the uniqueness of normal forms in the m-calculus (shown by the confluence of the m-calculus, Theorem 30 in Section 4) implies that the equations between translations of normalizing λ-terms in the m-calculus are conservative over the equations between the original terms in the λ-calculus. We conjecture but have not proved that this also holds for λ-terms without normal forms. We also conjecture that the encoding has PSN (preservation of strong normalization) w.r.t. the λ-calculus.

3.2 Component Renaming

An *atomic renaming* is a triple $f \overset{p}{:=} g$ of an old name f, a new name g, and a polarity $p \in \{+, -\}$, where p indicates whether an output $(p = +)$ or input $(p = -)$ component is to be renamed. A *renaming* is a finite set $r = \{\, f_i \overset{p_i}{:=} g_i \mid i = 1..n \,\}$ of atomic renamings such that for $1 \le i < j \le n$ if $f_i = f_j$ then $p_i = p_j = +$ and if $g_i = g_j$ then $p_i = p_j = -$. The statement $i = 1..n$ abbreviates $1 \le i \le n$. Such a renaming is *easy* if $\{\, f_i \mid i = 1..n \,\} \cap \{\, g_i \mid i = 1..n \,\} = \emptyset$ and is otherwise *hard*. Note that a renaming can duplicate an output component as in $\{f \overset{+}{:=} g, f \overset{+}{:=} h\}$ or merge multiple input components as in $\{f \overset{-}{:=} h, g \overset{-}{:=} h\}$.

Easy renamings are applied to m-calculus terms as follows. First, the positive and negative atomic renamings are mapped into m-calculus terms like this:

$$(f \overset{+}{:=} g) = \{f \rhd \mathrm{x} = \bullet,\ g \rhd \mathrm{y} = x\} \qquad\qquad (f \overset{-}{:=} g) = (g \overset{+}{:=} f)$$

Then, given an easy renaming $r = \{\, f_i \overset{p_i}{:=} g_i \mid i = 1..n \,\}$, the application of r to M is:

$$M[r] = ((f_1 \overset{p_1}{:=} g_1) \oplus \cdots \oplus (f_n \overset{p_n}{:=} g_n) \oplus M) \backslash \{\, f_i \mid 1 \le i \le n \,\}$$

It is slightly more complicated to apply hard renamings. The method uses a set of distinguished dummy component names $\{\mathrm{d}_1, \mathrm{d}_2, \ldots\}$ and only works correctly if the dummy component names are not used for any other purpose. Let $r = \{\, f_i \overset{p_i}{:=} g_i \mid i = 1..n \,\}$ be a difficult renaming. Let $\{f_1, \ldots, f_n\} = \{h_1, \ldots, h_m\}$ where for $1 \le i \le n$ it holds that $f_i = h_{j_i}$ and for $1 \le i < j \le m$ it holds that $h_i \ne h_j$. The renaming r is factored into r_{a} and r_{b} as follows:

$$\begin{aligned} r_{\mathrm{a}} &= \{\, f_i \overset{p_i}{:=} \mathrm{d}_{j_i} \mid i = 1..n \,\} \\ r_{\mathrm{b}} &= \{\, \mathrm{d}_{j_i} \overset{p_i}{:=} g_i \mid i = 1..n \,\} \end{aligned}$$

Then the application of r to M is defined as:

$$M[r] = M[r_\mathrm{a}][r_\mathrm{b}]$$

The notation defined here for renaming has some subtleties. Consider the application $M[r]$ of renaming $r = \{\, f_i \overset{+}{:=} f'_i \mid i = 1..n \,\} \cup \{\, g_i \overset{-}{:=} g'_i \mid i = 1..m \,\}$. If M rewrites to a module with output components that overlap with $\{\, g'_i \mid i = 1..m \,\}$, then this will serve to resolve the renamed input components with the output components of the same module. Similarly, if M rewrites to a module with input components that overlap with $\{\, f'_i \mid i = 1..m \,\}$, then the renamed output components will be resolved with the input components. We see this as a feature of the notation. If M rewrites to a module where $\{\, f_i \mid i = 1..n \,\}$ are not all output components or $\{\, g_i \mid i = 1..m \,\}$ are not all input components, then the notation does not work properly.

An example of renaming a component is as follows. Let $\mathrm{M} = \{\mathrm{f} = \mathrm{N},\ \mathrm{g} \triangleright \mathrm{y} = \bullet\}$ and consider how renaming the output component "f" of M works:

$$\begin{array}{lll}
 & & \mathrm{M}[\{\mathrm{f} \overset{+}{:=} \mathrm{h}\}] \\
 & =_\mathrm{s} & (\{\mathrm{f} = \mathrm{N},\ \mathrm{g} \triangleright \mathrm{y} = \bullet\} \oplus \{\mathrm{f} \triangleright \mathrm{x} = \bullet,\ \mathrm{h} = \mathrm{x}\})\backslash \mathrm{f} \\
 & =_\mathrm{s} & (\{\mathrm{f} \triangleright \mathrm{x} = \mathrm{N},\ \mathrm{g} \triangleright \mathrm{y} = \bullet\} \oplus \{\mathrm{f} \triangleright \mathrm{x} = \bullet,\ \mathrm{h} \triangleright \mathrm{z} = \mathrm{x}\})\backslash \mathrm{f} \\
 & & \quad \text{where } \{\mathrm{x}, \mathrm{z}\} \cap \mathrm{FV}(\mathrm{N}) = \emptyset \\
(\textbf{link}) & \longrightarrow & \{\mathrm{f} \triangleright \mathrm{x} = \mathrm{N},\ \mathrm{g} \triangleright \mathrm{y} = \bullet,\ \mathrm{h} \triangleright \mathrm{z} = \mathrm{x}\}\backslash \mathrm{f} \\
(\textbf{hide-present}) & \longrightarrow & \{_ \triangleright \mathrm{x} = \mathrm{N},\ \mathrm{g} \triangleright \mathrm{y} = \bullet,\ \mathrm{h} \triangleright \mathrm{z} = \mathrm{x}\} \\
(\textbf{subst}) & \longrightarrow & \{_ \triangleright \mathrm{x} = \mathrm{N},\ \mathrm{g} \triangleright \mathrm{y} = \bullet,\ \mathrm{h} \triangleright \mathrm{z} = \mathrm{N}\} \\
(\textbf{gc-module}) & \longrightarrow & \{\mathrm{g} \triangleright \mathrm{y} = \bullet,\ \mathrm{h} \triangleright \mathrm{z} = \mathrm{N}\} \\
 & =_\mathrm{s} & \{\mathrm{h} = \mathrm{N},\ \mathrm{g} \triangleright \mathrm{y} = \bullet\}
\end{array}$$

When there are cycles the results do not look as pretty although everything still works properly. If we consider instead $\mathrm{M}_1 = \{\mathrm{f} \triangleright \mathrm{x} = \mathrm{N}'(\mathrm{x}),\ \mathrm{g} \triangleright \mathrm{y} = \bullet\}$, where by $\mathrm{N}'(\mathrm{x})$ we mean that x occurs in N', then we can get at best the following:

$$\mathrm{M}_1[\{\mathrm{f} \overset{+}{:=} \mathrm{h}\}] \twoheadrightarrow \{\mathrm{h} = \mathrm{N}'(\mathrm{x}),\ \mathrm{x} = \mathrm{N}'(\mathrm{x}),\ \mathrm{g} \triangleright \mathrm{y} = \bullet\} = \mathrm{M}_2$$

The following non-rewriting is the case:

$$\mathrm{M}_1[\{\mathrm{f} \overset{+}{:=} \mathrm{h}\}] \not\twoheadrightarrow \{\mathrm{h} \triangleright \mathrm{x} = \mathrm{N}'(\mathrm{x}),\ \mathrm{g} \triangleright \mathrm{y} = \bullet\} = \mathrm{M}_3$$

The terms M_2 and M_3 are observationally equivalent but are not equated by the m-calculus.

3.3 Records and Record Operations

By the syntactic sugar in Figure 3, record syntax is already accepted by the m-calculus. Furthermore, the expected rewrite rule for selection is simulated:

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

Symmetric record concatenation is encoded as $(M_1 \oplus M_2)$. Asymmetric record concatenation where the right record takes precedence in the case of conflicts can be encoded using hiding, if the conflicting fields are known. For example, if field f is supplied by M_2 and might be supplied by M_1 and no other conflicts are possible, asymmetric concatenation can be encoded as $(M_1\backslash f) \oplus M_2$. (Handling cases where the conflicting fields are not known in advance would require extensions to the component hiding operator and/or to the linking operator. We leave this to future work.) Record extension can be encoded in a similar fashion.

3.4 Modules

C-style The m-calculus directly supports the modules of C-like languages. 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 start routine of the program is represented as $(P.\mathrm{main})$.

Package-style The m-calculus can also easily represent other styles of module systems. 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:

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

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:

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

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

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

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. For this it is convenient to add a small extension to the m-calculus. For a set of component names $\mathcal{F}$, extend the m-calculus with the operation $(M\backslash\overline{\mathcal{F}})$ that hides all components in module M except those listed in $\mathcal{F}$. (We have not reproved the results in Section 4 with this extension but we are confident the proofs are straightforward.) 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:

$$\begin{aligned}
\mathrm{M'_B} &= \left\{\begin{array}{l}\mathrm{A} \triangleright \mathrm{y} = \bullet,\ \mathrm{C} \triangleright \mathrm{z} = \bullet,\ \mathrm{B} \triangleright \mathrm{w} = \{\mathrm{i} = 10,\ \mathrm{g} = \mathrm{P(x.f, x.h, x.i)},\ \mathrm{f} = \mathrm{x.f}\},\\ \mathrm{x} = (\mathrm{y}\backslash\overline{\{\mathrm{f,h,i}\}}) \oplus (\mathrm{z}\backslash\overline{\{\mathrm{f,h,i}\}}) \oplus (\mathrm{w}\backslash\overline{\{\mathrm{f,h,i}\}})\end{array}\right\}\\
\mathrm{M_C} &= \{\mathrm{C} = \{\mathrm{h} = \mathrm{R}\}\}
\end{aligned}$$

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:

$$(\mathrm{M_A} \oplus \mathrm{M'_B} \oplus \mathrm{M_C} \oplus \mathrm{M_{Main}}).\mathrm{Main.main} \twoheadrightarrow \mathrm{Q}(\mathrm{N}, \mathrm{P}(\mathrm{N}, \mathrm{R}, 10), 5)$$

In the above example, observe that if $\mathrm{M'_B}$ is linked with two modules $\mathrm{M'_A}$ and $\mathrm{M'_C}$ whose A and C components both supply f, then the linking operation in $\mathrm{M'_B}$ 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. (We leave types to future work.) Such module systems provide modules called *structures* as well as a λ-calculus (*functors* and *functor applications*) for manipulating them. The level of modules is layered on top of the core language and the two levels can not be freely intermixed. A structure is essentially a dependent record; it is dependent in the sense that later fields can refer to the values of earlier fields. Mutually recursive dependencies between fields are allowed only in the case of core value definitions (which are given in the form `val rec` $pat_1 = exp_1$ `and ... and` $pat_n = exp_n$). Mutual recursion is not allowed at the module level. A functor is essentially a λ-abstractions 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.

The Calculus of Ancona and Zucca Following a long series of their earlier papers on modules and *mixins*, Ancona and Zucca have defined a calculus with primitive modules [AZ99]. In our discussion, we will call their calculus the AZ-calculus. The constructor in the AZ-calculus is the *module*, written $[\iota; o; \rho]$ where ι, o, and ρ are total functions with finite domains. The function ι maps (deferred) variables to (input) component names, the function o maps (output) component names to terms, and the function ρ maps (private) variables to terms. It is required that $\mathrm{dom}(\iota) \cap \mathrm{dom}(\rho) = \emptyset$. The pairs in ι, o, and ρ correspond respectively to m-calculus input, output, and private components. The other operators in the AZ-calculus are (1) *sum*, written $M + N$, (2) *reduct*, written ${}_{\sigma^\iota|}M_{|\sigma^o}$ where σ^ι and σ^o are renamings, (3) *freeze*, written $\mathit{freeze}_{\sigma^f}(M)$ where σ^f is a renaming, and (4) *selection*, written $M.f$. In addition, the AZ-calculus is (1) presented as being parameterized over an unspecified core calculus and (2) given a type system with simple (but not polymorphic) types, but we will not discuss either of these aspects of it.

In Ancona and Zucca's paper, they use capital letters for component names and lowercase letters for variables, but we will instead follow the syntactic conventions of this paper in presenting the AZ-calculus. They use the notation $X_i \xmapsto{i=1..n} Y_i$ to denote the function $\{ X_i \mapsto Y_i \mid i = 1..n \}$ with domain $\{ X_i \mid i = 1..n \}$ and codomain $\{ Y_i \mid i = 1..n \}$. Their presentation of renamings is quite different from the renamings we define in Section 3.2. In their presentation a renaming σ is a total function from a finite set of component names $\mathrm{dom}(\sigma)$ to another finite set of component names $\mathrm{cod}(\sigma)$, and can be non-surjective, having a finite difference between its codomain and range. They write a renaming as $\sigma = f_i \xmapsto{i \in I} g_i, g_j{}^{j \in J}$, denoting that $\sigma(f_i) = g_i$ for $i \in I$, $\mathrm{dom}(\sigma) = \{ f_i \mid i \in I \}$, $\mathrm{cod}(\sigma) = \{ g_i \mid i \in I \cup J \}$, and $\mathrm{ran}(\sigma) = \{ g_i \mid i \in I \}$. When this notation is used it is required that $\{ g_i \mid i \in I \} \cap \{ g_j \mid j \in J \} = \emptyset$.

The terms of the AZ-calculus can be encoded in the m-calculus as follows:

$$
\begin{aligned}
\iota &= (f^{\mathrm{i}}_1 \triangleright y_1 = \bullet), \ldots, (f^{\mathrm{i}}_m \triangleright y_m = \bullet), (\_ \triangleright x_1 = y_{k_1}), \ldots (\_ \triangleright x_n = y_{k_n}) \\
&\qquad \text{where } \iota = x_j \xmapsto{j=1..n} f_{k_j} \text{ and } \{f_{k_1}, \ldots, f_{k_n}\} = \{f_1, \ldots, f_m\} \\
&\qquad\qquad \text{and } f_i \neq f_j \text{ for } 1 \leq i < j \leq m \\
o &= (f^{\mathrm{o}}_1 \triangleright \_ = M_1), \ldots, (f^{\mathrm{o}}_n \triangleright \_ = M_n) \quad \text{where } o = f_j \xmapsto{j=1..n} M_j \\
\rho &= (\_ \triangleright x_1 = M_1), \ldots, (\_ \triangleright x_n = M_n) \quad \text{where } \rho = x_j \xmapsto{j=1..n} M_j \\
[\iota; o; \rho] &= \{\iota, o, \rho\} \\
M + N &= M \oplus N \\
\mathit{freeze}_{\sigma^f}(M) &= M[\{\, f^{\mathrm{i}}_j \mathrel{:\overset{-}{=}} g^{\mathrm{o}}_j \mid j \in J \,\}] \qquad\qquad \text{where } \sigma^f = f_j \xmapsto{j \in J} g_j, {g_k}^{k \in K} \\
{}_{\sigma^\iota|}M_{|\sigma^o} &= M[\{\, f^{\mathrm{i}}_j \mathrel{:\overset{-}{=}} g^{\mathrm{i}}_j \mid j \in J \,\} \cup \{\, g^{\mathrm{o}}_k \mathrel{:\overset{+}{=}} f^{\mathrm{o}}_k \mid k \in K \,\}] \backslash \{\, g^{\mathrm{o}}_l \mid l \in L \,\} \oplus \{g^{\mathrm{i}}_1 = \bullet, \ldots, g^{\mathrm{i}}_n = \bullet\} \\
&\qquad \text{where } \sigma^\iota = f_j \xmapsto{j \in J} g_j, {g_k}^{k=1..n} \text{ and } \sigma^o = f_k \xmapsto{k \in K} g_k, {g_l}^{l \in L}
\end{aligned}
$$

The AZ-calculus selection operator is translated as itself.

Note that input and output component names of the AZ-calculus are translated into disjoint subsets of CompName, i.e., a component name f becomes f^{i} if it is for an input component and becomes f^{o} if it is for an output component. This reflects the fact that input and output component names are effectively different namespaces in the AZ-calculus. The translation of the input map ι uses extra private components because ι is not necessarily injective. In translating the reduct operator, any non-surjectivity of the output component renaming σ^o is handled by the hiding operation and any non-surjectivity of the input component renaming σ^ι is handled by the linking with a module containing only dummy input components. Note that an output component renaming σ^o in the AZ-calculus is reversed from the way we define renaming of output components in Section 3.2. Our translation of the AZ-calculus freeze operator ignores any extra elements in the codomain of the renaming because their definition seems to ignore them also.

Although we have not proved this, we conjecture that the translation of the AZ-calculus is faithful in the sense that it preserves observable behavior. While their (sum) rewrite rule is simulated on the translations of AZ-calculus terms, their (reduct), (freeze), and (selection) rules are not simulated exactly. However, it is possible to see a correspondence by considering the infinite trees that result from unwinding bindings. In the m-calculus, these infinite trees can be thought of as the limit of applying a modified version of the (**subst**) rule arbitrarily many times. The modification to (**subst**) is to simply drop the side condition on dependency between components. A similar notion of infinite tree can be obtained for the AZ-calculus. If one then considers the effect of the rewrite rules on these infinite trees, it is possible to see that there are only inessential differences between the AZ-calculus rewrite steps on AZ-calculus terms and the m-calculus rewrite steps on the translations of the terms. We leave the formalization of this idea to future work. We have not investigated whether there is a faithful translation from the m-calculus into the AZ-calculus.

3.5 Objects (ς-Calculus)

Unsurprisingly, as the m-calculus supports encodings of the λ-calculus and records with field restriction (i.e., component hiding), the following record-of-methods encoding for the ς-calculus [AC96] works fine. We write "!" for the method invocation operator to avoid confusion with our component selection operator "." and we also use the previous λ-calculus encoding.

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

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

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

More interestingly, the following encoding (inspired by [BL92]) is also easy, using a distinguished component name "self".

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

The ς-calculus rewrite rules are also simulated using this encoding, although the encoding does not have PSN w.r.t. the ς-calculus. (A more complicated version of this encoding appears to have PSN.) The benefit of this encoding is that it allows a nice freezing operator to be defined. For example, to freeze the f method of an object:

$$\begin{aligned}
\text{freeze}_f\, M = {} & (\text{let } x = M \text{ in } ((x \oplus \{\text{self} = y \backslash f \oplus \{f = z\}, g \triangleright y = \bullet, f \triangleright z = \bullet\}) \backslash \text{self})[\{g :\bar{=} \text{self}\}]) \\
& \text{where } x,\ y,\ z, \text{ and } g \text{ are fresh}
\end{aligned}$$

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.

3.6 First-Class Contexts

Contexts are terms with holes where filling the hole of a context can capture variables. In [HO97], Hashimoto and Ohori extended the λ-calculus to manipulate contexts as first-class entities. They added *named holes*, written X, *hole abstraction* to form contexts, written $\delta X.M$, and *context application* (i.e., context filling), written $M_1 \odot M_2$.

Hashimoto and Ohori described in [HO97] the difficulties they had in designing their calculus. A very important goal for a calculus is that the meaning of an expression should be independent of the order in which the rewrite rules are applied. However, this goal seemed to conflict with the key feature of context-hole filling, the feature that distinguishes it from substituting for a variable: the capturing of variables by binders inside the context. The solution Hashimoto and Ohori devised involves (1) treating context application as a construct which binds variables in the argument (the term which will fill the hole in the context), (2) adding *variable renamers* which map ordinary variables into *interface variables* and *vice versa*, and (3) annotating context application and named holes with variable renamers.

In Hashimoto and Ohori's presentation, each variable renamer ν is an injective function from ordinary variables to ordinary variables with finite domain of definition, because they use the same set for ordinary variables and interface variables. However, interface variables are not subject to α-conversion so we will present them as a separate set of labels like record field names. We will also use more common notation for writing finite functions. We then see there are two kinds of variable renamers: named holes are annotated with renamers from interface variables to ordinary variables, i.e., X^ν where ν is of the form $\{f_1 \mapsto x_1, \ldots, f_n \mapsto x_n\}$, and context applications are annotated with renamers from ordinary variables to interface variables, i.e., $M_1 \odot_\nu M_2$ where ν is of the form $\{x_1 \mapsto f_1, \ldots, x_n \mapsto f_n\}$. An example rewrite step is:

$$(\delta X.\lambda x.(y\, X^{\{f \mapsto x\}})) \odot_{\{z \mapsto f\}} (yz) \longrightarrow \lambda x.y(yx)$$

Notice that the variable z is captured by the λ-binding for x when the context hole is filled in, because of the combination of the two variable renamers $\{z \mapsto f\}$ and $\{f \mapsto x\}$.

Hashimoto and Ohori's calculus can be easily encoded in the m-calculus. We extend the λ-calculus encoding given above as follows, associating with each named hole X an ordinary variable x_X:

$$\begin{aligned}
X^{\{f_1 \mapsto x_1, \ldots, f_n \mapsto x_n\}} &= (x_X \oplus \{f_1 = x_1,\ \ldots,\ f_n = x_n\}).\text{res} \\
(\delta X.M) &= (\lambda x_X.M) \\
\big(M_1 \odot_{\{x_1 \mapsto f_1, \ldots, x_n \mapsto f_n\}} M_2\big) &= M_1\{f_1 \triangleright x_1 = \bullet,\ \ldots,\ f_n \triangleright x_n = \bullet,\ \text{res} = M_2\}
\end{aligned}$$

This encoding faithfully simulates both α-equivalence and all rewrite sequences of Hashimoto and Ohori's calculus.

3.7 First-Class Environments

Sato, Sakurai, and Burstall's $\lambda\varepsilon$-calculus [SSB99, SB98] extends the λ-calculus by adding first-class *environments*, written $\{M_1/x_1, \ldots, M_n/x_n\}$, and *evaluation under an environment*, written $M_1[\![M_2]\!]$. In $\lambda\varepsilon$, the $M_1[\![M_2]\!]$ syntax is a binding construct. There are peculiarities about how this construct works in $\lambda\varepsilon$ which suggest alternate encodings. First, the set of variables which $M_1[\![M_2]\!]$ binds in the term M_2 depends on the type of M_1 (which must evaluate to an environment). The type specifies the variables for which M_1 supplies values. Thus, if types are erased from a $\lambda\varepsilon$-term, it becomes impossible to determine the term's meaning. Second, the variables that are bound in M_2 by $M_1[\![M_2]\!]$ are not subject to α-conversion although λ-bound variables are α-converted as usual. This is reminiscent of the usual distinction between variable names and record field names: the latter are not subject to α-conversion.

The meaning of $\lambda\varepsilon$-terms becomes clear when it is encoded using records as follows, associating a field name f_x with each variable x:

$$\begin{aligned}\{M_1/x_1,\ldots,M_n/x_n\} &= \{f_{x_1} = M_1,\ldots,f_{x_n} = M_n\}\\ M_1[\![M_2]\!] &= (\text{let } x = M_1 \text{ in } (M_2[x_1 := x.f_{x_1},\ \ldots,\ x_n := x.f_{x_n}]))\\ &\quad\text{where } x \text{ is fresh and } M_1\text{'s type is } \{x_1^{\tau_1},\ldots,x_n^{\tau_n}\}\end{aligned}$$

This record encoding makes clear why the variables bound by $M_1[\![M_2]\!]$ are not α-convertible: they disappear completely in the encoding. Unfortunately, this encoding is not structural in the sense that it does not simply build terms up. A structural encoding of the $\lambda\varepsilon$-calculus can be achieved in the m-calculus using a distinguished component name "res" as follows:

$$\begin{aligned}\{M_1/x_1,\ldots,M_n/x_n\} &= \{f_{x_1} = M_1,\ldots,f_{x_n} = M_n\}\\ M_1[\![M_2]\!] &= (M_1 \oplus \{f_{x_1} \triangleright x_1 = \bullet,\ \ldots,\ f_{x_n} \triangleright x_n = \bullet,\ \text{res} = M_2\}).\text{res}\\ &\quad\text{where } M_1\text{'s type is } \{x_1^{\tau_1},\ldots,x_n^{\tau_n}\}\end{aligned}$$

In this m-calculus encoding, the variable bindings of $M_1[\![M_2]\!]$ become explicit and also become α-convertible. Neither of these encodings preserve rewrite sequences of $\lambda\varepsilon$ but both capture the intended meaning of terms. Unlike the $\lambda\varepsilon$-calculus, neither encoding relies on types to give meaning to terms.

4 The Well-Behavedness of the Rewrite Rules

This section establishes not only that the m-calculus is *confluent* but also that it satisfies the *strong finite developments* property. To prove these results, we use a variation of the m-calculus which adds *redex marks* for tracking residuals of redexes. We show that a rewrite sequence in the marked m-calculus can be lowered to a rewrite sequence in the (unmarked) m-calculus. We also show that the marked m-calculus is confluent and strongly normalizing, thus establishing the *strong finite developments* property of the m-calculus. This implies the confluence of the m-calculus.

We first prove strong normalization of the marked m-calculus. Then, to prove confluence of the marked m-calculus we need only weak confluence on the terms (structural equivalence classes of raw terms). We prove weak confluence on the terms by proving an equivalent result on the raw terms. This means that our proof deals with and accounts for every structural operation (i.e., α-conversion and re-ordering) explicitly. We believe this level of detail is necessary to be confident in the result as the m-calculus contains binding features which are complicated in comparison with the λ-calculus.

4.1 Properties of Renaming

In the proofs that follow, we will rely on a number of properties of renaming. We prove them here.

Lemma 3 (Renaming Sanity)

1. $M[\![x := x]\!] = M$.
2. $M[\![x := y]\!] = M$ *if* $x \notin \mathrm{FV}(M)$.
3. $x \notin \mathrm{FV}(M[\![x := y]\!])$ *if* $y \notin \mathrm{Capture}_x(M)$ *and* $x \neq y$.
4. $M[\![x := y]\!][\![y := x]\!] = M$ *if* $y \notin \mathrm{FV}(M)$.

Points 3 and 4 essentially amount to correctness of renaming. Points 1 and 2 are stability results that we could not have hoped to obtain had we employed fresh-naming in the definition of renaming. (Remember that $\cdot[\![\cdot := \cdot]\!]$ is defined on raw terms.)

Proof:

1. By structural induction — observe that only identity variable replacements are possible.

2. By structural induction — observe that no variable replacements are possible.

3. By structural induction

   - We have:
   $$z[\![x := y]\!] = \begin{cases} z & \text{if } x \neq z \\ y & \text{otherwise} \end{cases}$$
   But, by assumption, $x \neq y$, hence $x \notin \mathrm{FV}(z[\![x := y]\!])$.
   - Considering $\{D\}[\![x := y]\!]$ we see that, in the first case, we are done by I.H. In the second case, either $x \in \mathrm{Binders}(D)$ and we are done, or $x \notin \mathrm{Binders}(D)$, but then $y \in \mathrm{Binders}(D)$ which implies that $x \notin \mathrm{FV}(D)$ by $y \notin \mathrm{Capture}_x(\{D\})$.
   - The cases $(M\backslash f)[\![x := y]\!], (M_1 \oplus M_2)[\![x := y]\!], (M.f)[\![x := y]\!]$ all follow directly by I.H.
   - The last case, $\langle M \mid D\rangle[\![x := y]\!]$, is parallel to the case for $\{D\}[\![x := y]\!]$.

4. By structural induction

   - We have:
   $$z[\![x := y]\!][\![y := x]\!] = \begin{cases} z & \text{if } y \neq z \\ x & \text{otherwise} \end{cases}$$
   But the side-condition on the statement we are proving, $y \notin \mathrm{FV}(M)$ with $M = y = z$, prohibits the latter possibility and we are done.
   - For $\{D\}[\![x := y]\!][\![y := x]\!]$, we are done by definition if $\{x, y\} \cap \mathrm{Binders}(D) \neq \emptyset$, otherwise we are done by I.H.
   - The cases $(M\backslash f)[\![x := y]\!][\![y := x]\!], (M_1 \oplus M_2)[\![x := y]\!][\![y := x]\!], (M.f)[\![x := y]\!][\![y := x]\!]$ all follow directly by I.H.
   - The last case, $\langle M \mid D\rangle[\![x := y]\!][\![y := x]\!]$, is parallel to the case for $\{D\}[\![x := y]\!][\![y := x]\!]$. □

4.2 The Marked m-Calculus

$$\begin{array}{rcllll}
r & \in & \{0, 1, 2, \ldots\} & & & \text{(redex marks)} \\
\bar{c} & ::= & (F \triangleright x = \bar{B}) \mid (_ \triangleright x = \bar{M})^0 & & & \\
\bar{D} & ::= & \bar{c}_1, \ldots, \bar{c}_n \quad \text{where } n \geq 0 & & & \\
\bar{B} & ::= & \bar{M} \mid \bullet & & & \\
\bar{M} & ::= & x & \mid \; x^r & \mid & \\
 & & \{\bar{D}\} & \mid & & \\
 & & (\bar{M}\backslash f) & \mid \; (\{\bar{D}\}\backslash^0 f) & \mid & \\
 & & (\bar{M}_1 \oplus \bar{M}_2) & \mid \; (\{\bar{D}_1\} \oplus^0 \{\bar{D}_2\}) & \mid & \\
 & & (\bar{M}.f) & \mid \; (\{\bar{D}\}.^0 f) & \mid & \\
 & & \langle \bar{M} \mid \bar{D}\rangle & \mid \; \langle\{\bar{D}_1\} \mid \bar{D}_2\rangle^0 & \mid & \langle \bar{M} \mid \rangle^0
\end{array}$$

Figure 9: The preterms of the marked m-calculus: PreTerm′, extending PreTerm.

The marked preterms of the m-calculus are defined in Figure 9. We will use R as a meta-variable over blank and the redex marks such that, e.g., $(\bar{M}\backslash^R f)$ means either $(\bar{M}\backslash f)$ or $(\bar{M}\backslash^0 f)$. Various operations and terminology from Section 2 are extended to work with marked preterms in the straightforward way. This includes the operations defined in Figures 2, 4, 5, and 6 as well as the operations defined in the upper parts of Figures 7 and 8.

Definition 4 (Nicely Marked Preterms) *A mark r occurring in a marked preterm $\bar{M}$ participates in a computational redex up to α-conversion iff r is matched by the left-hand side of one of the rules in Figure 10, ignoring the following side conditions of the rules:*

- *The requirements in (**link′**) that $\bar{D}[-\mathcal{F}]$ and $\bar{D}'[-\mathcal{F}]$ do not clash on variable names and that these do not end up capturing free variables stemming from the other module and that $\bar{D}[\mathcal{F}]$ and $\bar{D}'[\mathcal{F}]$ overlap on binder names.*
- *The requirements in (**subst′**) and (**subst-letrec′**) concerning captured variables.*

$$
\begin{array}{ll}
\textbf{(link}'\textbf{)} & \{\bar{D}\} \oplus^0 \{\bar{D}'\} \rightsquigarrow' \{\bar{D}[-\mathcal{F}], \bar{D}'[-\mathcal{F}], \bar{D}''\} \\
& \quad \text{where } \begin{cases} \mathcal{F} = \{f_1, \ldots, f_n\} = \text{Names}(\bar{D}) \cap \text{Names}(\bar{D}'), \\ \text{Binders}(\bar{D}[-\mathcal{F}]) \cap (\text{Binders}(\bar{D}') \cup \text{FV}(\bar{D}')) = \emptyset, \\ \text{Binders}(\bar{D}'[-\mathcal{F}]) \cap (\text{Binders}(\bar{D}) \cup \text{FV}(\bar{D})) = \emptyset, \\ \bar{D}[\mathcal{F}] \ = (f_1 \triangleright x_1 = \bar{B}_1), \ldots, (f_n \triangleright x_n = \bar{B}_n), \\ \bar{D}'[\mathcal{F}] = (f_1 \triangleright x_1 = \bar{B}'_1), \ldots, (f_n \triangleright x_n = \bar{B}'_n), \\ \bar{D}'' \quad = (f_1 \triangleright x_1 = \bar{B}''_1), \ldots, (f_n \triangleright x_n = \bar{B}''_n), \\ \bar{B}''_i = \text{PickBody}(\bar{B}_i, \bar{B}'_i) \text{ is defined for } 1 \leq i \leq n \end{cases} \\
\textbf{(subst}'\textbf{)} & \bar{D} \rightsquigarrow' \bar{D}[i := (F_i \triangleright x_i = \bar{C}[\bar{M}_j])^R] \\
& \quad \text{where } \begin{cases} \bar{D}[i] = (F_i \triangleright x_i = \bar{C}[{x_j}^r])^R, \\ \bar{D}[j] = (F_j \triangleright x_j = \bar{M}_j)^{R'}, \\ \text{Capture}_\Box(\bar{C}) \cap (\{x_j\} \cup \text{FV}(\bar{M}_j)) = \emptyset, \\ \text{not DependsOn}_{\bar{D}}(j, i) \end{cases} \\
\textbf{(subst-letrec}'\textbf{)} & \langle \bar{C}[x^r] \mid \bar{D} \rangle \rightsquigarrow' \langle \bar{C}[\bar{M}] \mid \bar{D} \rangle \\
& \quad \text{where } \begin{cases} \bar{D}[i] = (\_ \triangleright x = \bar{M})^R \text{ for some } i, \\ \text{Capture}_\Box(\bar{C}) \cap (\{x\} \cup \text{FV}(\bar{M})) = \emptyset \end{cases} \\
\textbf{(select}'\textbf{)} & \{\bar{D}\}.^0 f \rightsquigarrow' \langle x_i \mid \bar{D}' \rangle \\
& \quad \text{where } \begin{cases} \bar{D} \ = (F_1 \triangleright x_1 = \bar{M}_1)^{R_1}, \ldots, (f \triangleright x_i = \bar{M}_i), \ldots, (F_n \triangleright x_n = \bar{M}_n)^{R_n}, \\ \bar{D}' = (\_ \triangleright x_1 = \bar{M}_1)^{R_1}, \ldots, \ (\_ \triangleright x_i = \bar{M}_i), \ldots, (\_ \triangleright x_n = \bar{M}_n)^{R_n} \end{cases} \\
\textbf{(gc-module}'\textbf{)} & \{\bar{D}\} \rightsquigarrow' \{\bar{D}[I]\} \\
& \quad \text{where } \begin{cases} I, J \text{ partition } \{1, \ldots, |\bar{D}|\}, \\ J \neq \emptyset, \\ \text{Binders}(\bar{D}[J]) \cap \text{FV}(\bar{D}[I]) = \emptyset, \\ \text{for } j \in J \text{ it holds that } \bar{D}[j] = (\_ \triangleright x_j = \bar{M}_j)^0 \end{cases} \\
\textbf{(gc-letrec}'\textbf{)} & \langle \bar{M} \mid \bar{D} \rangle^R \rightsquigarrow' \langle \bar{M} \mid \bar{D}[I] \rangle^R \\
& \quad \text{where } \begin{cases} I, J \text{ partition } \{1, \ldots, |\bar{D}|\}, \\ J \neq \emptyset, \\ \text{Binders}(\bar{D}[J]) \cap \big(\text{FV}(\bar{M}) \cup \text{FV}(\bar{D}[I])\big) = \emptyset, \\ \text{for } j \in J \text{ it holds that } \bar{D}[j] = (\_ \triangleright x_j = \bar{M}_j)^0 \end{cases} \\
\textbf{(empty-letrec}'\textbf{)} & \langle \bar{M} \mid \rangle^0 \rightsquigarrow' \bar{M} \\
\textbf{(closure}'\textbf{)} & \langle \{\bar{D}\} \mid \bar{D}' \rangle^0 \rightsquigarrow' \{\bar{D}, \bar{D}'\} \\
& \quad \text{where } \begin{cases} |\bar{D}'| > 0, \\ \text{Binders}(\bar{D}) \cap (\text{Binders}(\bar{D}') \cup \text{FV}(\bar{D}')) = \emptyset \end{cases} \\
\textbf{(hide-present}'\textbf{)} & \{\bar{D}[i := (f \triangleright x = \bar{M})]\} \backslash^0 f \rightsquigarrow' \{\bar{D}[i := (\_ \triangleright x = \bar{M})]\} \\
\textbf{(hide-absent}'\textbf{)} & \{\bar{D}\} \backslash^0 f \rightsquigarrow' \{\bar{D}\} \\
& \quad \text{where } f \notin \text{Labels}(\bar{D})
\end{array}
$$

Figure 10: The marked computational rules.

- *The requirements in (**closure′**) that binders do not overlap and that the binders of $\bar{D}$ do not capture free variables of $\bar{D}'$.*

A marked preterm $\bar{M}$ is nicely marked iff (1) every mark in $\bar{M}$ participates in a computational redex up to α-conversion and (2) every marked variable occurrence has a mark greater than any of the marks in the component body (which can not be $\bullet$) paired with the corresponding binder.

The set of marked raw terms, RawTerm′, is defined like RawTerm with the additional constraint that every marked raw term is a nicely marked preterm. For marked raw terms the structural contraction relation $\rightsquigarrow'_{\mathrm{s}}$, the structural rewrite relation $\dashrightarrow'_{\mathrm{s}}$, and the structural equivalence relation $='_{\mathrm{s}}$ are the straightforward extensions of $\rightsquigarrow_{\mathrm{s}}$, $\dashrightarrow_{\mathrm{s}}$, and $=_{\mathrm{s}}$ to marked terms. The marked computational contraction relation $\rightsquigarrow'_{\mathrm{c}}$ is the union of the contraction relations defined by the rules in Figure 10. This differs from the unmarked version as follows:

- In (**link′**), (**select′**), (**empty-letrec′**), (**closure′**), (**hide-present′**), and (**hide-absent′**), the disappearing syntax constructor has to be marked.
- In the substitution rules, (**subst′**) and (**subst-letrec′**), the variable occurrence being replaced has to be marked.
- In the garbage collection rules, (**gc-letrec′**) and (**gc-module′**), all of the disappearing module components have to be marked.

The marked computational rewrite relation $\dashrightarrow'_{\mathrm{c}}$ and the rewrite relation on marked raw terms $\dashrightarrow'$ are obtained as in the unmarked case.

Lemma 5 (Closure under Marked Rewriting) *If $\bar{M}$ is a marked raw term and $\bar{M} \dashrightarrow' \bar{N}$, then $\bar{N}$ is a marked raw term.*

Proof: We saw in Lemma 1 that $\dashrightarrow \subseteq \mathrm{RawTerm} \times \mathrm{RawTerm}$ and, as this extends to the marked calculus up-to marking by definition, we need only prove that nice marking is preserved.

- Binding is preserved and any weight is non-increasing. That is, a variable occurrence remains bound by the "same" component (under a suitable notion of sameness). The body of a component can be changed by (**link′**), (**subst′**), and (**subst-letrec′**). In the former case, normal occurrences of the binding variable are unmarked. In the latter two cases, the weight of the body decreases by the nicely marked-condition on variables.
- No mark ends up not designating a redex up-to α under contraction. The proof is by a case analysis on the marks. It is itemized by the contraction rule a mark designates.
  - (**link′**): Straightforwardly no changes possible to D and D' invalidating the considered contraction (up-to α).
  - (**subst′**), (**subst-letrec′**): Garbage collection can remove the considered substitution source. But, by the side-condition on no overlap of free and bound variables in the garbage collection rules, would such a garbage collection necessarily remove the considered substitution target as well. As for the $\mathrm{DependsOn}_D$-condition: we need to argue that no two components in a module that are not related can become related after some contraction. Two such components can only become related by the addition of new components to the module ((**link′**) and (**closure′**)) or by extension of existing components ((**subst′**) and (**subst-letrec′**)) but not by their removal ((**gc-module′**) and (**gc-letrec′**)). In the case of (**subst′**) and (**subst-letrec′**), the $\mathrm{DependsOn}_D$ is preserved by its transitive closure. For (**link′**) and (**closure′**), no new dependencies are possible by the required non-overlap of bound and free variables of the components in question. Observe that, in the case of (**link′**), new dependencies can be introduced between components from the different modules but that does not lead to new dependencies between components stemming from the same module by the second clause in the definition of $\mathrm{DependsOn}_D$.
  - (**select′**): No changes possible to D invalidating the considered contraction as garbage collection cannot remove components with a name proper.
  - (**gc-letrec′**), (**gc-module′**): Substitution can insert new, free variables into a body in I. However, neither of these can be binders in components sought garbage collected by the side-condition on no overlap of free and binding variables in the garbage collection rules.

– (**empty-letrec′**): No changes possible to M invalidating the considered contraction.

– (**closure′**): All of D' can be garbage collected in which case the mark designates an (**empty-letrec′**)-redex.

– (**hide-present′**): No changes possible to D invalidating the considered contraction as garbage collection cannot remove components with a name proper.

– (**hide-absent′**): Straightforwardly no changes possible to D invalidating the considered contraction.

□

Definition 6 (The Marked m-Calculus) *The marked m-calculus is* $\mathcal{M}' = (\text{Term}', \longrightarrow') = (\text{RawTerm}', \dashrightarrow'_{\text{c}})/ =_{\text{s}}'$*. (See Definition 2.)*

Definition 7 (Mark Erasure) *If* $\bar{M}$ *is a marked raw term, then* $\lfloor \bar{M} \rfloor$ *is the raw term that is the result of removing the marks from* $\bar{M}$*.*

Before we present the next result we need to introduce some notation.

Notation 8 (Commuting Diagrams) *In diagrams, vertexes written* $\bullet$ *or* $\circ$ *range over entities of the types appropriate for the edges which connect to them. Edges in diagrams denote binary predicates on the vertexes they connect. Vertex annotations denote unary predicates on the vertexes to which they are attached. Generally, the vertexes denote terms and the edges denote membership in rewriting relations. Diagrams come in two flavours.*

1. *When there are no light edges, then the diagram is simply an assertion that the indicated predicate uses are true. For example, the following diagram simply states that* $\bar{M} \dashrightarrow' \bar{N}$*:*

$$\bar{M} \overset{\prime}{\dashrightarrow} \bar{N}$$

2. *When there are both dark and light edges, then the diagram is a quantified statement that should be read as "for every assignment of entities to the dark vertexes satisfying their vertex annotations such that the predicates indicated by dark edges are satisfied, there exists an assignment of entities to the light vertexes satisfying their vertex annotations such that the predicates indicated by the light edges are true". For example, the diagram*

$$\begin{tikzcd} & \bullet \arrow[dl] \arrow[dr] & \\ \bullet \arrow[dr] & & \bullet\ (\mathrm{P}) \arrow[dl] \\ & \circ\ (\mathrm{Q}) & \end{tikzcd}$$

is equivalent to the statement

$$\forall M, N_1, N_2.\ (M \to N_1 \wedge M \to N_2 \wedge P(N_2)) \Rightarrow \exists N.\ (N_1 \to N \wedge N_2 \to N \wedge Q(N))$$

Lemma 9 *For any (unmarked) raw terms,* M_1 *and* M_2*, where* $M_1 \dashrightarrow M_2$*, there exist marked raw terms,* $\bar{M}_1$ *and* $\bar{M}_2$*, such that* $\lfloor \bar{M}_i \rfloor = M_i$ *for* $i \in \{1, 2\}$ *and* $\bar{M}_1 \dashrightarrow' \bar{M}_2$*. Conversely, for any marked raw terms,* $\bar{N}_1$ *and* $\bar{N}_2$*, where* $\bar{N}_1 \dashrightarrow' \bar{N}_2$ *we have that* $\lfloor \bar{N}_1 \rfloor \dashrightarrow \lfloor \bar{N}_2 \rfloor$*:*

$$\begin{tikzcd} \bullet \arrow[r, dashed] & \bullet \\ \circ \arrow[u, "\lfloor - \rfloor"] \arrow[r, dashed, "\prime"] & \circ \arrow[u, "\lfloor - \rfloor"'] \end{tikzcd} \qquad \begin{tikzcd} \bullet \arrow[r, dashed] & \bullet \\ \bullet \arrow[u, "\lfloor - \rfloor"] \arrow[r, dashed, "\prime"] & \bullet \arrow[u, "\lfloor - \rfloor"'] \end{tikzcd}$$

Proof: In the first case, we can simply put 0's on the pieces of syntax forming the redex in accordance with the correspondence between marks and contractions. The second case follows from the definition of marked contraction as unmarked contraction with superimposed marks. □

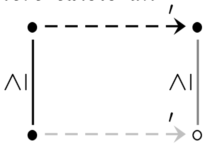
4.3 Strong Normalization of the Marked m-Calculus

The technique used here to prove the strong normalization of the marked m-calculus is very similar to a technique used in [AK97].

Definition 10 (Weight) *The weight of a marked raw term is the multiset of all of the redex marks occurring in it.*

Definition 11 *Given two marked raw terms, $\bar{M}$ and $\bar{N}$, $\bar{M}$ is* more marked than $\bar{N}$, *written $\bar{M} \geq \bar{N}$, iff $\lfloor \bar{M} \rfloor = \lfloor \bar{N} \rfloor$ and for each mark occurring in $\bar{N}$ there is a (possibly different) mark at the corresponding position in $\bar{M}$.*

Lemma 12 *Given $\bar{M} \geq \bar{N}$ and $\bar{N} \dashrightarrow' \bar{N}'$, there exists an $\bar{M}'$ such that $\bar{M} \dashrightarrow' \bar{M}'$ and $\bar{M}' \geq \bar{N}'$:*



Proof: $\bar{N}$ marked reduces to $\bar{N}'$ by contraction of a sub-term of $\bar{N}$. Going via $\lfloor \bar{M} \rfloor = \lfloor \bar{N} \rfloor$ we can find a sub-term in $\bar{M}$ corresponding to it. By definition of $\geq$ will this always form a redex. Contract it. □

Lemma 13 *For any $M \in$ Term there is at least one $\bar{M} \in$ Term$'$ maximal w.r.t. $\geq$ such that $\lfloor \bar{M} \rfloor = M$. (There is either one such maximal marked term or an infinite number of them, depending on whether there is a substitution redex.)*

Proof: A maximal amount of 0-marks designating all contractions, but the substitutions, are easily put on any raw term. With regards to the substitutions can we observe that they are *acyclic* due to the side-condition preventing dependencies. For any module, $\{\bar{D}\}$, and letrec, $\langle \bar{M} \mid \bar{D} \rangle^R$, with $\bar{D}$ and $\bar{M}$ maximally annotated with marks, we can proceed as follows: compute the DAG (i.e., directed acyclic graph) of dependencies between the components, find the weight of the bodies at the bottom level (the ones that do not depend on others that do not, at the same time, depend on them), annotate a mark one higher than the biggest mark in the weight of the respective body (0 if it is empty) to free occurrences of the binding variables (at the lowest level) in bodies at strictly higher levels, proceed upwards (to M in the case of a letrec). Do not annotate marks to free occurrences of variables binding $\bullet$. □

Lemma 14 (Strong Normalisation of $\longrightarrow'$) *There are no infinite sequences of $\longrightarrow'$.*

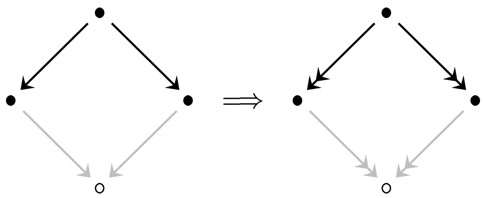
Proof: Marked structural rewriting preserves weight while marked computational rewriting decreases weight relative to the standard multiset ordering. □

This result explains why only variable occurrences need redex labels greater than 0: all other marks are removed and not replaced by others.

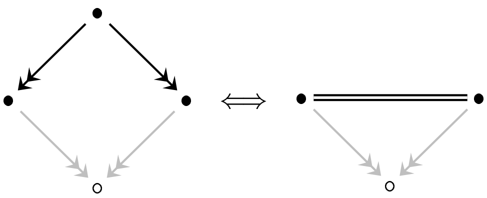
4.4 A Few Classical Results

Before we start on the actual results we remind the reader of the following well known facts.

Lemma 15 (Hindley-Rosen) *The Diamond Property for any relation $\rightarrow$ implies the Diamond Property for its transitive, reflexive closure, $\twoheadrightarrow$, a.k.a. confluence:*



Proposition 16 (Confluence and Church-Rosser are Equivalent) *For any relation $\rightarrow$, its transitive, reflexive closure, $\twoheadrightarrow$, and its transitive, reflexive, and symmetric closure, $=$, we have:*



Lemma 17 (Newman [New42]) *Any weakly confluent and strongly normalizing relation is also confluent.*

4.5 Confluence of the Marked m-Calculus

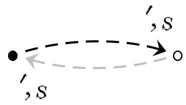
The following notion, akin to the widely used bound-variable convention, turns out to be helpful at various points in the ensuing technical developments.

Definition 18 (Nice Structural Forms) *A marked raw term $\bar{M}$ is a* nice structural form *(NSF) iff (1) all binders in $\bar{M}$ use names not in* $\mathrm{FV}(\bar{M})$ *and (2) all binders use names distinct from each other, except that binders in modules connected by a marked linking operator are required to share names as necessary to enable the (**link**′) rule.*

Lemma 19 *Each structural equivalence class contains at least one NSF. (It will contain exactly one if the term has no binders and otherwise will have an infinite number.)*

Proof: Pick as many fresh variables as there are bound variables in any marked raw term in consideration (minus those outlined in the statement of the proposition). Now use the α-rules one by one from the bottom and up. □

Lemma 20 ($\dashrightarrow'_{\mathrm{s}}$ 1-step Reversibility)



Proof: We first observe that it suffices to prove the result for $\rightsquigarrow'_{\mathrm{s}}$, the structural contraction relation.

- (**comp-order**′) and (**link-commute**′) are straightforwardly invertible.
- For (α-**module**′) and (α-**letrec**′) we can assume that $x \neq y$ as, otherwise, we are done by Lemma 3 (1). Now, consider without loss of generality:

  $$\langle \bar{M} \mid \bar{D} \rangle^R \rightsquigarrow' \langle \bar{M}[\![x := y]\!] \mid \bar{D}' \rangle^R$$

  where

  $$y \notin \mathrm{UnSafeNewNames}(x, \langle \bar{M} \mid \bar{D} \rangle^R)$$

  If we can justify:

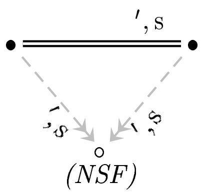
  $$x \notin \mathrm{UnSafeNewNames}(y, \langle \bar{M}[\![x := y]\!] \mid \bar{D}' \rangle^R)$$

  we are done by Lemma 3 (4).

  - We have $x \notin \mathrm{Binder}(\bar{D}')$ by definition of the raw terms and the fact that $x \in \mathrm{Binder}(\bar{D})$ was changed by the first contraction.
  - We have $x \notin \mathrm{FV}(\bar{M}[\![x := y]\!]) \cup \mathrm{FV}(\bar{D}')$ from Lemma 3 (3) by choice of y in the first contraction.
  - Finally, we obtain $x \notin \mathrm{Capture}_y(\bar{M}[\![x := y]\!]) \cup \bigcup_{1 \leq i \leq |\bar{D}'|} \mathrm{Capture}_x(\mathrm{Body}(\bar{D}'[i]))$ by assuming the contrary. Then either, say, $x \in \mathrm{FV}(\bar{M}[\![x := y]\!])$ or x is being bound in, say, $\bar{M}[\![x := y]\!]$ with an occurrence of y (free in $\bar{M}[\![x := y]\!]$) in its scope. The first case is impossible by Lemma 3 (3), the second by choice of y.□

As we hinted after Lemma 3, this result essentially amounts to correctness of the structural, and in particular the α-conversion, rules in the sense that the result shows that binding is properly preserved. One can read the result to say that the original binding can be restored after an α-contraction in a conservative manner: as determined by syntactic identity. By extension this can only happen if correct binding was never lost.

Lemma 21 (Church-Rosser of $\dashrightarrow'_{\mathrm{s}}$)



Proof: By the Hindley-Rosen Lemma and Lemma 16 on the diamond property of $\dashrightarrow'_{\mathrm{s}}$, a result which can be obtained directly from Lemma 20. The closing vertex can always be brought into NSF by Lemma 19. □

Lemma 22 (Weak Commutativity of $\dashrightarrow'_{\mathrm{s}}$ and $\dashrightarrow'_{\mathrm{c}}$)

$$\begin{tikzcd}
 & \bullet \arrow[dl, dashed, "{\prime,\mathrm{c}}"'] \arrow[dr, dashed, "{\prime,\mathrm{s}}"] & \\
\bullet \arrow[dr, dashed, gray, "{\prime,\mathrm{s}}"'] & & \bullet\ \textit{(NSF)} \arrow[dl, dashed, gray, "{\prime,\mathrm{c}}"] \\
 & \circ &
\end{tikzcd}$$

Proof: See Appendix B. The proof relies heavily on the right-hand vertex being a NSF. □

Lemma 23 (Weak Confluence of $\dashrightarrow'_{\mathrm{c}}$)

$$\begin{tikzcd}
 & \bullet\ \textit{(NSF)} \arrow[dl, dashed, "{\prime,\mathrm{c}}"'] \arrow[dr, dashed, "{\prime,\mathrm{c}}"] & \\
\bullet \arrow[dr, dashed, gray, "{\prime,\mathrm{c}}"'] & & \bullet \arrow[dl, dashed, gray, "{\prime,\mathrm{c}}"] \\
 & \circ &
\end{tikzcd}$$

Proof: See Appendix C. The proof relies to a limited extent on the top vertex being a NSF. □

Lemma 24 (Weak Confluence of $\dashrightarrow'_{\mathrm{c}}$ on $='_{\mathrm{s}}$-representatives)

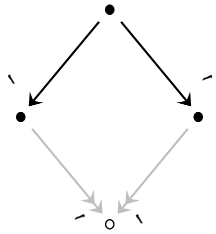
$$\begin{tikzcd}
\bullet \arrow[rr, equal, "{\prime,\mathrm{s}}"'] \arrow[d, dashed, "{\prime,\mathrm{c}}"'] & & \bullet \arrow[d, dashed, "{\prime,\mathrm{c}}"] \\
\bullet \arrow[dr, dashed, gray, "\prime"'] & & \bullet \arrow[dl, dashed, gray, "\prime"] \\
 & \circ &
\end{tikzcd}$$

Proof: Given marked raw terms $\bar{M}_1$, $\bar{M}_2$, $\bar{N}_1$, & $\bar{N}_2$, we have a NSF $\bar{M}$ and marked raw terms $\bar{N}'_1$, $\bar{N}'_2$, & $\bar{N}$ such that:

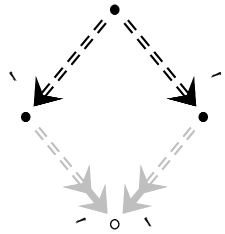
$$\begin{tikzcd}
\bar{M}_1 \arrow[rrrr, equal, "{\prime,\mathrm{s}}"'] \arrow[d, dashed, "{\prime,\mathrm{c}}"'] \arrow[drr, dashed, "{\prime,\mathrm{s}}"'] & & \text{Lemma 21} & & \bar{M}_2 \arrow[d, dashed, "{\prime,\mathrm{c}}"] \arrow[dll, dashed, "{\prime,\mathrm{s}}"] \\
\bar{N}_1 \arrow[dr, dashed, "{\prime,\mathrm{s}}"'] & \text{Lemma 22} & \bar{M}\ (NSF) \arrow[dl, dashed, "{\prime,\mathrm{c}}"] \arrow[dr, dashed, "{\prime,\mathrm{c}}"'] & \text{Lemma 22} & \bar{N}_2 \arrow[dl, dashed, "{\prime,\mathrm{s}}"] \\
 & \bar{N}'_1 \arrow[dr, dashed, "{\prime,\mathrm{c}}"'] & \text{Lemma 23} & \bar{N}'_2 \arrow[dl, dashed, "{\prime,\mathrm{c}}"] & \\
 & & \bar{N} & &
\end{tikzcd}$$

□

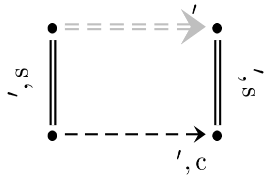
Lemma 25 (Weak Confluence of $\longrightarrow'$)



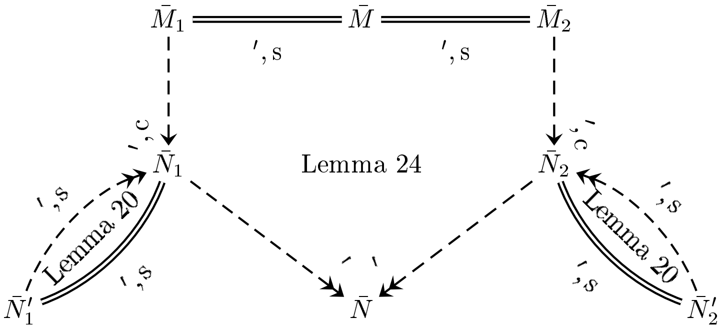
Proof: The considered diagram is definitionally equal to:



where $\Rightarrow' = (=_{\mathrm{s}}'; \dashrightarrow_{\mathrm{c}}'; =_{\mathrm{s}}')$ is the least relation, such that:

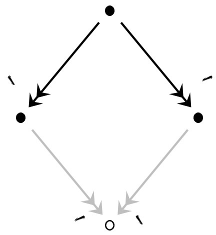


The desired result follows from Lemma 24 by repeated use of Lemma 20:



□

Lemma 26 (Confluence of $\longrightarrow'$)



Proof: From Lemma 25 & Lemma 14 by Newman's Lemma. □

4.6 Finite Developments and Confluence for the m-Calculus

Lemma 27 (Uniquely Existing Normal Forms for $\longrightarrow'$) *Every $='$-equivalence class has a unique $\longrightarrow'$-normal form which every term in the class rewrites to.*

Proof: By Lemma 26 and Lemma 14. □

Definition 28 (Developments) *Let* $\longrightarrow_{\mathrm{dev}} \subseteq \mathrm{Term} \times \mathrm{Term}$ *be the least relation such that:*

$$\forall \bar{M}_1, \bar{M}_2 \in \mathrm{Term}' \,.\, \bar{M}_1 \dashrightarrow\!\!\!\!\rightarrow' \bar{M}_2 \Rightarrow \lfloor \bar{M}_1 \rfloor \longrightarrow_{\mathrm{dev}} \lfloor \bar{M}_2 \rfloor$$

or, alternatively:

$$\begin{tikzcd} \bullet \arrow[r, dashed, two heads, "\prime"] \arrow[d, mapsto, "\lfloor - \rfloor"'] & \bullet \arrow[d, mapsto, "\lfloor - \rfloor"] \\ \bullet \arrow[r, gray, "\mathrm{dev}"'] & \bullet \end{tikzcd}$$

Lemma 29 (Diamond Property of $\longrightarrow_{\mathrm{dev}}$)

$$\begin{tikzcd} & \bullet \arrow[dl, "\mathrm{dev}"'] \arrow[dr, "\mathrm{dev}"] & \\ \bullet \arrow[dr, gray, "\mathrm{dev}"'] & & \bullet \arrow[dl, gray, "\mathrm{dev}"] \\ & \circ & \end{tikzcd}$$

Proof: Assume we have M, N_1, and N_2 where $M \longrightarrow_{\mathrm{dev}} N_i$. By Definition 28, Lemma 12, and Lemma 13 do we have $\bar{M}$, $\bar{N}_1$, and $\bar{N}_2$ such that $\lfloor \bar{M} \rfloor = M$, $\lfloor \bar{N}_i \rfloor = N_i$, and $\bar{M} \dashrightarrow\!\!\!\!\rightarrow' \bar{N}_i$. By Lemma 27 is there, therefore, an $\bar{N}$ such that $\bar{N}_i \dashrightarrow\!\!\!\!\rightarrow' \bar{N}$ and $\bar{M} \dashrightarrow\!\!\!\!\rightarrow' \bar{N}$:

$$\begin{tikzcd} & M \arrow[dl, "\mathrm{dev}"'] \arrow[dr, "\mathrm{dev}"] \arrow[dd, "\mathrm{dev}"'] & \\ N_1 \arrow[dr, "\mathrm{dev}"'] & & N_2 \arrow[dl, "\mathrm{dev}"] \\ & \lfloor \bar{N} \rfloor & \end{tikzcd}$$

□

Theorem 30 (Confluence of $\longrightarrow$)

$$\begin{tikzcd} & \bullet \arrow[dl, two heads] \arrow[dr, two heads] & \\ \bullet \arrow[dr, gray, two heads] & & \bullet \arrow[dl, gray, two heads] \\ & \circ & \end{tikzcd}$$

Proof: We are seen to have $\longrightarrow \subseteq \longrightarrow_{\mathrm{dev}} \subseteq \twoheadrightarrow$ by Lemma 9 and Definition 28. Therefore, the same relationship holds for the transitive, reflexive closures of these relations and we have $\twoheadrightarrow_{\mathrm{dev}} = \twoheadrightarrow$. From the Hindley-Rosen Lemma applied to Lemma 29 we obtain the Diamond Property of $\twoheadrightarrow_{\mathrm{dev}}$ which, in turn, reads "confluence of $\longrightarrow$" by the previous observation. □

Corollary 31 (Confluence of $\dashrightarrow$)

$$\begin{tikzcd} & \bullet \arrow[dl, dashed, two heads] \arrow[dr, dashed, two heads] & \\ \bullet \arrow[dr, gray, dashed, two heads] & & \bullet \arrow[dl, gray, dashed, two heads] \\ & \circ & \end{tikzcd}$$

An implication of Corollary 31 is that the user of the m-calculus can freely choose to work with either raw terms and $\dashrightarrow$ or with (real) terms (i.e., structural equivalence classes) and $\longrightarrow$. Both relations are confluent.

5 Related Work

5.1 Calculi with Linking

There has recently been quite a bit of work analyzing the meaning of linking. Cardelli presents a simply-typed calculus of linking for outermost-only modules without recursion [Car97]. Drossopoulou, Eisenbach, and Wragg give a module calculus tailored to reasoning about the quirks of the Java language definition [DEW99]. Ancona and Zucca give a calculus of primitive modules ("mixin modules") and linking which is quite similar to ours, but they choose to focus on the simply typed version of it rather than proving properties of the rewriting relation [AZ99]. Earlier, Ancona and Zucca also presented an algebra which can simplify module expressions, but is not powerful enough to represent general computation [AZ98].

From a point of view less oriented towards rewriting calculi, Flatt and Felleisen give a calculus of modules with similar capabilities to ours [FF98b]. Glew and Morrisett present a module calculus tailored towards dealing with linking of object files containing assembly-language-level code [GM99]. Waddell and Dybvig show how to encode a powerful module system with linking using the macro system of Scheme [WD99].

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 conflicting (but not in their system) components [DS96, DS98]. Bracha and Lindstrom present ways of encoding mixins using λ-calculus, records, and fix-point operators [BL92, Bra92]. Findler and Flatt describe how to use mixins and incomplete modules in actual programming [FF98a]. Flatt and Krishnamurthi and Felleisen present a calculus with an operational semantics for mixins and classes in the context of Java [FKF98].

5.3 Calculi for Cycles

Inspiring a large part of our formulation, Ariola and Klop did ground-breaking work on reasoning about λ-terms combined with a cycle-forming construct for mutually recursive definitions [AK97]. Ariola and Blom refined this work to deal with proving consistency in the absence of confluence [AB97a, AB97b].

5.4 ML-style Modules vs. Types

Much of the work on ML-style modules has focused on typing issues. For ML-style modules, relevant work includes that of Crary, Harper, and Puri in extending the ML module system to deal with recursion [CHP99]. Earlier work to add first-class modules (i.e., higher-order functors) to ML includes that of Russo [Rus98], Harper and Lillibridge [HL94, Lil97], and Leroy [Ler94]. An important work in proving the decidability of type checking for the ML module system was the work of Harper, Mitchell, and Moggi in reasoning about the phase distinction [HMM90]. Our work will eventually have to deal with this issue. In a slightly different vein than the work just mentioned, Jones showed that much of the complexity of typing ML-style modules could be avoided by allowing higher-order (parametric) signatures [Jon96, Jon95].

5.5 Types vs. Concatenation and Extension for Records and Objects

In extending our system to deal with types, it will probably be very helpful to consider the extensive literature on handling types for record concatenation [Wan89, HP91, Rém93b, Rém95], types for extensible records [Rém93a, Car93, Rém92, GJ96], and types for extensible objects [FHM94, RS9X, BBDCL9X].

References

[AB97a] Zena M. Ariola and Stefan Blom. Cyclic lambda calculi. In *Theoretical Aspects Comput. Softw. : Int'l Conf.*, 1997.

[AB97b] Zena M. Ariola and Stefan Blom. Lambda calculi plus letrec. Submitted, 3 July 1997.

[AC96] Martìn Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AK97] Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Inf. & Comput.*, 139:154–233, 1997.

[AZ98] Davide Ancona and Elena Zucca. An algebra of mixin modules. In F. Parisi 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.

[AZ99] Davide Ancona and Elena 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 September – 1 October 1999. Springer-Verlag.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[Bar96] J. G. P. Barnes. *Programming in Ada 95.* Addison-Wesley, 1996.

[BBDCL9X] Viviana Bono, Michele Bugliesi, Mariangiola Dezani-Ciancaglini, and Luigi Liquori. Subtyping constraints for incomplete objects. *Fundamenta Informaticae*, 199X. To appear.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. Int'l Conf. Computer Languages*, pages 282–290, 1992.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity, and Multiple Inheritance.* PhD thesis, Univ. of Utah, March 1992.

[Car93] Luca Cardelli. Extensible records in a pure calculus of subtyping. In Gunter and Mitchell [GM93].

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Conf. Rec. POPL '97: 24th ACM Symp. Principles of Prog. Languages*, 1997.

[CHP99] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *Proc. ACM SIGPLAN '99 Conf. Prog. Language Design & Implementation*, 1999.

[DEW99] Sophia Drossopoulou, Susan Eisenbach, and David 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.

[DS96] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proc. 1996 Int'l Conf. Functional Programming*, 1996.

[DS98] Dominic Duggan and Constantinos Sourelis. Parameterized modules, recursive modules, and mixin modules. In *ACM SIGPLAN Workshop on ML and its Applications*, 1998.

[FF98a] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proc. 1998 Int'l Conf. Functional Programming*, 1998.

[FF98b] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN '98 Conf. Prog. Language Design & Implementation*, 1998.

[FHM94] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conf. Rec. POPL '98: 25th ACM Symp. Principles of Prog. Languages*, 1998.

[GJ96] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Univ. of Nottingham, 1996.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[GM93] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects Of Object-Oriented Programming: Types, Semantics and Language Design.* MIT Press, 1993.

[GM99] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In POPL '99 [POPL99].

[Har91] S. P. Harbison. *Modula-3.* Prentice Hall, 1991.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL '94 [POPL94], pages 123–137.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Conf. Rec. 17th Ann. ACM Symp. Principles of Programming Languages*, 1990.

[HO97] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. Technical Report RIMS-1098, Research Inst. for Math. Sci., Kyoto University, 25 September 1997.

[HP91] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. Technical Report CMU-CS-90-157R, Carnegie Mellon Univ., 2 July 1991.

[Jon95] Mark 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.

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. In *Conf. Rec. POPL '96: 23rd ACM Symp. Principles of Prog. Languages*, 1996.

[Jon99] Haskell 98: A non-strict, purely functional language. Technical report, The Haskell 98 Committee, 1 February 1999. Currently available at http://haskell.org.

[KF98] Shriram Krishnamurthi and Matthias Felleisen. Toward a formal theory of extensible software. In *Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November 1998.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In POPL '94 [POPL94], pages 109–122.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, Carnegie Mellon Univ., May 1997.

[MTHM90] Robin Milner, Mads Tofte, Robert Harper, and David B. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1990.

[New42] M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Math.*, 43(2):223–243, 1942.

[POPL94] *Conf. Rec. 21st Ann. ACM Symp. Principles of Programming Languages*, 1994.

[POPL99] *Conf. Rec. POPL '99: 26th ACM Symp. Principles of Prog. Languages*, 1999.

[Rém92] Didier Rémy. Projective ML. In *Proc. 1992 ACM Conf. LISP Funct. Program.*, 1992.

[Rém93a] Didier Rémy. Type inference for records in a natural extension of ML. In Gunter and Mitchell [GM93].

[Rém93b] Didier Rémy. Typing record concatenation for free. In Gunter and Mitchell [GM93].

[Rém95] Didier Rémy. A case study of typechecking with constrained types: Typing record concatenation. Presented at the workshop on Advances in Types for Computer Science at the Newton Institute, Cambridge, UK, August 1995.

[RS9X] Jon G. Riecke and Christopher A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 199X. To appear.

[Rus98] Claudio V. Russo. *Types for Modules.* PhD thesis, Univ. of Edinburgh, 1998.

[SB98] Masahiko Sato and Rod Burstall. Explicit environments. In *First Int'l Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs*, Tsukuba, Japan, March 1998.

[SSB99] Masahiko Sato, Takafumi Sakurai, and Rod Burstall. Explicit environments. In *Fourth Int'l Conf. Typed Lambda Calculi and Applications*, L'Aquila, Italy, 7–9 April 1999.

[Wan89] Mitchell 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.

[WD99] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In POPL '99 [POPL99].

A Relative Positions of two Redexes

In this appendix, we will briefly outline the case-splitting employed for the proofs of Lemmas 22 and 23

Let $\mathbf{P}$ range over a not further specified notion over paths in (marked) raw terms. We will use $<$ for the (strict) prefix relation on paths, $=$ for the identity relation, and $\perp$ for the complement of $< \cup =$, that is, for two *diverging* paths. For a context, $\bar{C}$, and a raw term, $\bar{M}$, $\mathbf{P}_{\bar{C}\leadsto\bar{M}}$ is the path in the raw term $\bar{C}[\bar{M}]$ leading to the occurrence of $\bar{M}$ in question.

Consider $\bar{M} = \bar{C_1}[\bar{M_1}] = \bar{C_2}[\bar{M_2}]$ with $\bar{M_1} \leadsto'_A \bar{N_1}$ and $\bar{M_2} \leadsto'_B \bar{N_2}$. And let us try to close the ensuing diagram with C and D reductions:

$$\begin{tikzcd}
 & \bar{M} \arrow[dl, dashed, "\leadsto'_A"'] \arrow[dr, dashed, "\leadsto'_B"] & \\
\bar{C_1}[\bar{N_1}] \arrow[dr, dashed, "\leadsto'_C"'] & & \bar{C_2}[\bar{N_2}] \arrow[dl, dashed, "\leadsto'_D"] \\
 & ? &
\end{tikzcd}$$

The following situations are possible:

1. $\bar{C_1}[\bar{N_1}] = \bar{C_2}[\bar{N_2}]$ — we will implicitly assume this to not be the case in the other cases.
2. $\bar{C_1} = \bar{C_2}$, implying $\bar{M_1} = \bar{M_2}$.
3. $\mathbf{P}_{\bar{C_1}\leadsto\bar{M_1}} \perp \mathbf{P}_{\bar{C_2}\leadsto\bar{M_2}}$
4. $\mathbf{P}_{\bar{C_1}\leadsto\bar{M_1}} < \mathbf{P}_{\bar{C_2}\leadsto\bar{M_2}}$
5. $\mathbf{P}_{\bar{C_2}\leadsto\bar{M_2}} < \mathbf{P}_{\bar{C_1}\leadsto\bar{M_1}}$

The diagram of case 1 is not diverging. The diagram of case 3 is trivially closed when $A \subseteq D$ and $B \subseteq C$ and the reduction relation is contextually closed, as is the case here. The diagrams of 4 and 5 are symmetric when $A = B$, and hence not in Lemma 22. All in all, the actual proof burden for Lemma 22 comprises cases 2, 4, and 5 while for Lemma 23, it comprises cases 2 and 4.

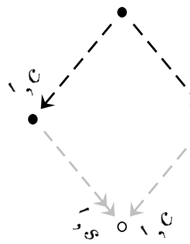
To conduct the proofs we will refer to reductions by a *name* of a contraction and the path leading to the term being contracted unless it is given by the context. Contraction names are given as the name of the contracting rule possibly juxtaposed with additional information to uniquely identify contractions (relative to specific raw terms).

$$\begin{aligned} cn \quad ::= \quad & (\alpha\textbf{-letrec}'){:}x_i, y \mid (\alpha\textbf{-module}'){:}x_i, y \mid (\textbf{comp-order}'){:}|D_1|, |D_1, c_1, D_2| \mid (\textbf{link-commute}') \mid \\ & (\textbf{link}') \mid (\textbf{subst}'){:}i, \mathbf{P}_{C\leadsto x_j} \mid (\textbf{subst-letrec}'){:}\mathbf{P}_{C\leadsto x} \mid (\textbf{select}') \mid (\textbf{gc-letrec}'){:}J \mid \\ & (\textbf{gc-module}'){:}J \mid (\textbf{empty-letrec}') \mid (\textbf{closure}') \mid (\textbf{hide-present}') \mid (\textbf{hide-absent}') \end{aligned}$$

B Proof of Lemma 22

We refer to Appendix A for an analysis of the proof burden and an outline of the employed case splitting and notation.

The proof is by induction on the length of the upper right leg. The base case, with length 0, is trivial. If, in turn, we can prove:



we are done as we, by induction hypothesis, can close any diagram with shorter (relative to the induction step) upper-right leg extending downwards to the right. Only problem is that some structural reductions, in fact, do turn computational redexes into a non-redexes.

When the right-hand vertex is a NSF we are safe but such an assumption is not available for the induction step. Instead, we have to strengthen the induction assumptions, targeting not only a NSF but a NSF where all bound variables are different from the bound and free variables of the originating term. By the proof of Lemma 19 is this a safe assumption.

That extra assumption allows us to restrict attention to Nice Structural Reductions in the upper-right leg of the above diagram and the induction goes through.

Definition 32 (Nice Structural Reduction) *A structural reduction is* nice*, called a NSR and denoted* $\dashrightarrow'_{\mathrm{NSR}}$*, if it is*

- *a* (**comp-order**$'$)
- *a* (**link-commute**$'$)
- *an* (α-**letrec**$'$) *and the new variable is fresh with respect to the term in consideration and the proper subterms of the contractum (the right-hand component of the contraction relation) are NSFs*
- *an* (α-**module**$'$) *and the new variable is fresh with respect to the term in consideration and the proper subterms of the contractum are NSFs*

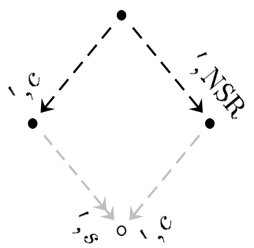
Excepted are α-reductions where the redex forms part of a (**link**$'$)*-redex up-to α. In that case we α-reduce two times the number of identically named components number of times.*

Proposition 33 *Any computational redex in a marked raw term that reduces to another marked raw term via a NSR, corresponds in a one-to-one manner to a computational redex of the same rule in the structural reduct (the right-hand component of the structural reduction relation).*

Proof: Straightforward. □

We see, by the preceding discussion, that it suffices to prove the following result.

Lemma 34



Lemma 5 ensures that any mark designates a redex, hence it suffices to argue for the presence of a mark to show that a particular redex is enabled up-to α.

Proof:

Case 2 : Case-splitting according to $\dashrightarrow'_{\mathrm{NSR}}$

Case (comp-order′) : Consider, e.g., the case with substitution (with the obvious abuse of notation)

$$\begin{tikzcd}[column sep=tiny]
& \bar{C}'[\{\bar{D}_1, (F \triangleright x = \bar{C}[x_j^r])^R, \bar{D}_2, \bar{c}_2, \bar{D}_3\}] \arrow[dl, dashed, "(\mathbf{subst'}):i, \mathbf{P}_{\bar{C} \rightsquigarrow x_j^r}"'] \arrow[dr, dashed, "(\mathbf{comp\text{-}order'})"] & \\
\bar{C}'[\{\bar{D}_1, (F \triangleright x = \bar{C}[\bar{M}])^R, \bar{D}_2, \bar{c}_2, \bar{D}_3\}] \arrow[dr, dashed, "(\mathbf{comp\text{-}order'})"'] & & \bar{C}'[\{\bar{D}_1, \bar{c}_2, \bar{D}_2, (F \triangleright x = \bar{C}[x_j^r])^R, \bar{D}_3\}] \arrow[dl, dashed, "(\mathbf{subst'}):i', \mathbf{P}_{\bar{C} \rightsquigarrow x_j^r}"] \\
& \bar{C}'[\{\bar{D}_1, \bar{c}_2, \bar{D}_2, (F \triangleright x = \bar{C}[\bar{M}])^R, \bar{D}_3\}] &
\end{tikzcd}$$

Case (link-commute′) :

$$\begin{tikzcd}
& \bar{C}[\{\bar{D}\} \oplus^0 \{\bar{D}'\}] \arrow[dl, dashed, "(\mathbf{link'})"'] \arrow[dr, dashed, "(\mathbf{link\text{-}commute'})"] & \\
\bar{C}[\{\bar{D}[-\mathcal{F}], \bar{D}'[-\mathcal{F}], \bar{D}''\}] \arrow[dr, dashed, "(\mathbf{comp\text{-}order'})"'] & & \bar{C}[\{\bar{D}'\} \oplus^0 \{\bar{D}\}] \arrow[dl, dashed, "(\mathbf{link'})"] \\
& \bar{C}[\{\bar{D}'[-\mathcal{F}], \bar{D}[-\mathcal{F}], \bar{D}''\}] &
\end{tikzcd}$$

Case (α-letrec′) : Consider, e.g., the case with $(\mathbf{gc\text{-}letrec'})\!:\!J$ where we garbage collect the component of the renamed binder

$$\begin{tikzcd}
& \bar{C}[\langle \bar{M} \mid \bar{D}, \bar{D}', (_ \triangleright x = \bar{B})^0\rangle^R] \arrow[dl, dashed, "(\mathbf{gc\text{-}letrec'}):J"'] \arrow[dr, dashed, "(\alpha\text{-}\mathbf{letrec'}):x,y"] & \\
\bar{C}[\langle \bar{M} \mid \bar{D}\rangle^R] & & \bar{C}[\langle \bar{M} \mid \bar{D}, \bar{D}'[\![x := y]\!], (_ \triangleright y = \bar{B}[\![x := y]\!])^0\rangle^R] \arrow[ll, dashed, "(\mathbf{gc\text{-}letrec'}):J"]
\end{tikzcd}$$

By the side-condition on the (**gc-letrec′**)-rule and Lemma 3 (2) do we have that $\bar{M}[\![x := y]\!] = \bar{M}$ and $\bar{D}[\![x := y]\!] = \bar{D}$.

Case (α-module′) : Consider the one interesting case where the (α-**module′**)-redex forms part of a (**link′**)-redex

$$\begin{tikzcd}
& \bar{C}[\{\bar{D}_1\} \oplus^0 \{\bar{D}_2\}] \arrow[dl, dashed, "(\mathbf{link'})"'] \arrow[dr, dashed, "NSR"] & \\
\bar{C}[\{\bar{D}_1[-\mathcal{F}], \bar{D}_2[-\mathcal{F}], \bar{D}'\}] \arrow[dr, dashed, "(\alpha\text{-}\mathbf{module'}):\overline{x},\overline{y}"'] & & \bar{C}[\{\bar{D}_1'\} \oplus^0 \{\bar{D}_2'\}] \arrow[dl, dashed, "(\mathbf{link'})"] \\
& \bar{C}[\{\bar{D}_1'[-\mathcal{F}], \bar{D}_2'[-\mathcal{F}], \bar{D}''\}] &
\end{tikzcd}$$

Case 4 : Computational contraction is outermost, the contraction(s) of the NSR are innermost. The cases are straightforward so we will instead present a non-example.

$$\begin{tikzcd}[column sep=small]
& \bar{C}[\langle\{(F \triangleright x = x)^R\} \mid (_ \triangleright y = y)^{R'}\rangle^0] \arrow[dl, dashed, "(\mathbf{closure}')"'] \arrow[dr, dashed, "(\alpha\text{-}\mathbf{module}'){:}x{,}y"] & \\
\bar{C}[\{(F \triangleright x = x)^R, (_ \triangleright y = y)^{R'}\}] & & \bar{C}[\langle\{(F \triangleright y = y)^R\} \mid (_ \triangleright y = y)^{R'}\rangle^0] \\
& ? &
\end{tikzcd}$$

Now, the right vertex is stuck with respect to (**closure′**) without a preceding α-conversion of the outermost y and that is not allowed by the desired result. Observe, however, that the (α-**module′**)-reduction is not a NSR as y is not fresh with respect to $\bar{C}[\langle\{(F \triangleright x = x)^R\} \mid (\_ \triangleright y = y)^{R'}\rangle^0]$ although it is a proper reduction.

Case 5 : The contraction(s) of the NSR are outermost, the computational contraction is innermost. The cases are again straightforward. Consider, e.g.,

$$\begin{tikzcd}[column sep=tiny]
& \bar{C}[\langle \bar{M} \mid (F \triangleright x = \{\bar{D}_1\} \oplus^0 \{\bar{D}_2\})^R, \bar{D}\rangle^R] \arrow[dl, dashed, "(\mathbf{link}')"'] \arrow[dr, dashed, "(\alpha\text{-}\mathbf{letrec}'){:}x{,}y"] & \\
\bar{C}[\langle \bar{M} \mid (F \triangleright x = \{\bar{D}_1[-\mathcal{F}], \bar{D}_2[-\mathcal{F}], \bar{D}''\})^R, \bar{D}\rangle^R] \arrow[dr, dashed, "(\alpha\text{-}\mathbf{letrec}'){:}x{,}y"'] & & \bar{C}[\langle \bar{M}' \mid (F \triangleright y = \{\bar{D}_1'\} \oplus^0 \{\bar{D}_2'\})^R, \bar{D}'\rangle^R] \arrow[dl, dashed, "(\mathbf{link}')"] \\
& \bar{C}[\langle \bar{M}' \mid (F \triangleright x = \{\bar{D}_1'[-\mathcal{F}], \bar{D}_2'[-\mathcal{F}], \bar{D}'''\})^R, \bar{D}'\rangle^R] &
\end{tikzcd}$$

□

C Proof of Lemma 23

We refer to Appendix A for an analysis of the proof burden and an outline of the employed case splitting and notation.

Lemma 5 ensures that any mark designates a redex, hence it suffices to argue for the presence of a mark to show that a particular redex is enabled up-to α. As we start out from a Nice Structural Form, we have, in fact, that all redexes are enabled without having to perform α-conversion first. The assumption that the top vertex is a NSF is used in only one place (substitution-substitution). The specific property used is that all bound variables are different. Non-overlap between bound and free variables (a.k.a. Barendregt's Variable Convention) is explicitly encoded into the rewrite rules and does not play a rôle in the proof. We will point out when the NSF assumption is used.

We present a non-complete selection of cases covering non-trivial as well as generic situations.

Case 2 :

- Upper-left leg is (**link′**) — no divergence possible.
- Upper-left leg is (**subst′**)$:i, \mathbf{P}_{\bar{C} \rightsquigarrow x_j^r}$ and upper-right leg is one of
  - (**subst′**)$:i', \mathbf{P}_{\bar{C}' \rightsquigarrow x_{j'}^{r'}}$:

Case $i = i'$ and (by assumption) $\bar{C} \neq \bar{C}'$: As DependsOn_ is transitively closed, we have that $\mathrm{DependsOn}_{\bar{D}} \supseteq \mathrm{DependsOn}_{\bar{D}[i:=\bar{C}[M_j]]} \cup \mathrm{DependsOn}_{\bar{D}[i:=\bar{C}'[M_{j'}]]}$. Consequently, we have an M (with "both substitutions performed"), such that

$$\begin{tikzcd}
& \bar{C}''[\bar{D}] \arrow[dl, dashed, "(\mathbf{subst'}):i,\mathbf{P}_{\bar{C}\rightsquigarrow x_j^r}"'] \arrow[dr, dashed, "(\mathbf{subst'}):i,\mathbf{P}_{\bar{C}'\rightsquigarrow x_{j'}^{r'}}"] & \\
\bar{C}''[\bar{D}[i := \bar{C}[M_j]]] \arrow[dr, dashed, "(\mathbf{subst'}):i,\mathbf{P}_{\bar{C}''''\rightsquigarrow x_{j'}^{r'}}"'] & & \bar{C}''[\bar{D}[i := \bar{C}'[M_{j'}]]] \arrow[dl, dashed, "(\mathbf{subst'}):j,\mathbf{P}_{\bar{C}'''\rightsquigarrow x_j^r}"] \\
& \bar{C}''[\bar{D}[i := M]] &
\end{tikzcd}$$

In future cases we will, indeed, rely on Lemma 5, rather than explicitly presenting the preceding argumentation, for the validity of the employed reductions.

Case $i \neq i'$ and $i \neq j'$ and $i' \neq j$:

$$\begin{tikzcd}
& \bar{C}''[\bar{D}] \arrow[dl, dashed, "(\mathbf{subst'}):i,\mathbf{P}_{\bar{C}\rightsquigarrow x_j^r}"'] \arrow[dr, dashed, "(\mathbf{subst'}):i',\mathbf{P}_{\bar{C}'\rightsquigarrow x_{j'}^{r'}}"] & \\
\bar{C}''[\bar{D}[i := \bar{C}[M_j]]] \arrow[dr, dashed, "(\mathbf{subst'}):i',\mathbf{P}_{\bar{C}'\rightsquigarrow x_{j'}^{r'}}"'] & & \bar{C}''[\bar{D}[i' := \bar{C}'[M_{j'}]]] \arrow[dl, dashed, "(\mathbf{subst'}):i,\mathbf{P}_{\bar{C}\rightsquigarrow x_j^r}"] \\
& \bar{C}''[\bar{D}[i := \bar{C}[M_j]][i' := \bar{C}'[M_{j'}]]] &
\end{tikzcd}$$

Case $i \neq i'$ and $i \neq j'$ and $i' = j$ (WLOG, also $i \neq i'$ and $i = j'$ and $i' \neq j$) :

$$\begin{tikzcd}
& \bar{C}''[\bar{D}] \arrow[dl, dashed, "(\mathbf{subst'}):i,\mathbf{P}_{\bar{C}\rightsquigarrow x_j^r}"'] \arrow[dr, dashed, "(\mathbf{subst'}):i',\mathbf{P}_{\bar{C}'\rightsquigarrow x_{j'}^{r'}}"] & \\
\bar{C}''[\bar{D}[i := \bar{C}[\bar{C}'[x_{j'}^{r'}]]]] \arrow[d, dashed, "(\mathbf{subst'}):i,\mathbf{P}_{\bar{C}[\bar{C}']\rightsquigarrow x_{j'}^{r'}}"] & & \bar{C}''[\bar{D}[i' := \bar{C}'[M_{j'}]]] \arrow[ddl, dashed, "(\mathbf{subst'}):i,\mathbf{P}_{\bar{C}\rightsquigarrow x_j^r}"] \\
\bar{C}''[\bar{D}[i := \bar{C}[\bar{C}'[\bar{M}_{j'}]]]] \arrow[dr, dashed, "(\mathbf{subst'}):i',\mathbf{P}_{\bar{C}'\rightsquigarrow x_{j'}^{r'}}"'] & & \\
& \bar{C}''[\bar{D}[i := \bar{C}[\bar{C}'[\bar{M}_{j'}]]][i' := \bar{C}'[M_{j'}]]] &
\end{tikzcd}$$

For the middle-left leg, no free variables of $\bar{M}_{j'}$ are being captured by binders in $\bar{C}$ as $\bar{C}''[\bar{D}]$ is a NSF.

Case $i \neq i'$ and $i = j'$ and $i' = j$: Not possible due to "not $\mathrm{DependsOn}_{\bar{D}}(j, i)$" condition in rule.

– (**subst-letrec**$'$)$:\mathbf{P}_{\bar{C}'\leadsto x^{r'}}$: Parallel to the cases where $i \neq i'$ above.

– (**gc-module**$'$)$:J$: We have two cases where we assume WLOG that J and I partitions the component collection into $\bar{D}$ and $\bar{D}'$, respectively.

Case $i \in J$:

$$\begin{tikzcd}
 & \bar{C}'[\bar{D},\bar{D}'] \arrow[dl, dashed, "(\mathbf{subst}')\,:\,i{,}\,\mathbf{P}_{\bar{C}\leadsto x_j^r}"'] \arrow[dr, dashed, "(\mathbf{gc\text{-}module}')\,:\,J"] & \\
\bar{C}'[\bar{D}[i := \bar{C}[M_j]],\bar{D}'] \arrow[rr, dashed, "(\mathbf{gc\text{-}module}')\,:\,J"'] & & \bar{C}'[\bar{D}']
\end{tikzcd}$$

Case $i \notin J$: By side-conditions on the (**gc-module**$'$)-rule do we have that also $j \notin J$.

$$\begin{tikzcd}
 & \bar{C}'[\bar{D},\bar{D}'] \arrow[dl, dashed, "(\mathbf{subst}')\,:\,i+|D|{,}\,\mathbf{P}_{\bar{C}\leadsto x_{j+|D|}^r}"'] \arrow[dr, dashed, "(\mathbf{gc\text{-}module}')\,:\,J"] & \\
\bar{C}'[\bar{D},\bar{D}'[i := \bar{C}[M_j]]] \arrow[dr, dashed, "(\mathbf{gc\text{-}module}')\,:\,J"'] & & \bar{C}'[\bar{D}'] \arrow[dl, dashed, "(\mathbf{subst}')\,:\,i{,}\,\mathbf{P}_{\bar{C}\leadsto x_j^r}"] \\
 & \bar{C}'[\bar{D}'[i := \bar{C}[M_j]]] &
\end{tikzcd}$$

– (**gc-letrec**$'$)$:J$: Parallel to the above case.

– (**closure**$'$):

$$\begin{tikzcd}
 & \bar{C}[\langle\{\bar{D}\} \mid \bar{D}'\rangle^0] \arrow[dl, dashed, "(\mathbf{subst}')\,:\,i{,}\,\mathbf{P}_{\bar{C}\leadsto x_j^r}"'] \arrow[dr, dashed, "(\mathbf{closure}')"] & \\
\bar{C}[\langle\{\bar{D}\} \mid \bar{D}'[i := \bar{M}]\rangle^0] \arrow[dr, dashed, "(\mathbf{closure}')"'] & & \bar{C}[\{\bar{D},\bar{D}'\}] \arrow[dl, dashed, "(\mathbf{subst}')\,:\,i+|D|{,}\,\mathbf{P}_{\bar{C}\leadsto x_{j+|D|}^r}"] \\
 & \bar{C}[\{\bar{D},\bar{D}'[i := \bar{M}]\}] &
\end{tikzcd}$$

- Upper-left leg is (**subst-letrec**$'$) : $\mathbf{P}_{\bar{C}\leadsto x^r}$ and upper-right leg is one of (**subst-letrec**$'$) : $\mathbf{P}_{\bar{C}'\leadsto x'^{r'}}$, (**gc-letrec**$'$)$:J$, or (**closure**$'$): Parallel to the relevant sub-cases of upper-left leg a (**subst**$'$)$:i,\mathbf{P}_{\bar{C}\leadsto x_j^r}$.
- Upper-left leg is (**select**$'$) — no divergence possible.
- Upper-left leg is (**gc-module**$'$) : J and upper-right leg is (**gc-module**$'$) : J'. It suffices to

consider the following situation:

$$\begin{tikzcd}
 & \bar{C}[\bar{D}, \bar{D}', \bar{D}''] \arrow[dl, dashed, "(\mathbf{gc\text{-}module'})\!:J"'] \arrow[dr, dashed, "(\mathbf{gc\text{-}module'})\!:J'"] & \\
\bar{C}[\bar{D}, \bar{D}'] \arrow[dr, dashed, "(\mathbf{gc\text{-}module'})\!:J'"'] & & \bar{C}[\bar{D}, \bar{D}''] \arrow[dl, dashed, "(\mathbf{gc\text{-}module'})\!:J"] \\
 & \bar{C}[\bar{D}] &
\end{tikzcd}$$

- Upper-left leg is $(\mathbf{gc\text{-}letrec'})\!:J$ and upper-right leg is one of
  - $(\mathbf{gc\text{-}letrec'})\!:J'$: Parallel to above case.
  - $(\mathbf{closure'})$:
    Case $I = \emptyset$:

$$\begin{tikzcd}
 & \bar{C}[\langle\{\bar{D}\} \mid \bar{D}'\rangle^0] \arrow[dl, dashed, "(\mathbf{gc\text{-}letrec'})\!:J"'] \arrow[dr, dashed, "(\mathbf{closure'})"] & \\
\bar{C}[\langle\{\bar{D}\} \mid \rangle^0] \arrow[dr, dashed, "(\mathbf{empty\text{-}letrec'})"'] & & \bar{C}[\{\bar{D}, \bar{D}'\}] \arrow[dl, dashed, "(\mathbf{gc\text{-}module'})\!:J"] \\
 & \bar{C}[\{\bar{D}\}] &
\end{tikzcd}$$

    Case $I \neq \emptyset$:

$$\begin{tikzcd}
 & \bar{C}[\langle\{\bar{D}\} \mid \bar{D}', \bar{D}''\rangle^0] \arrow[dl, dashed, "(\mathbf{gc\text{-}letrec'})\!:J"'] \arrow[dr, dashed, "(\mathbf{closure'})"] & \\
\bar{C}[\langle\{\bar{D}\} \mid \bar{D}''\rangle^0] \arrow[dr, dashed, "(\mathbf{closure'})"'] & & \bar{C}[\{\bar{D}, \bar{D}', \bar{D}''\}] \arrow[dl, dashed, "(\mathbf{gc\text{-}module'})\!:J"] \\
 & \bar{C}[\{\bar{D}, \bar{D}''\}] &
\end{tikzcd}$$

- Upper-left leg is $(\mathbf{empty\text{-}letrec'})$ — no divergence possible.
- Upper-left leg is $(\mathbf{closure'})$ — no divergence possible.
- Upper-left leg is $(\mathbf{hide\text{-}present'})$ — no divergence possible.
- Upper-left leg is $(\mathbf{hide\text{-}absent'})$ — no divergence possible.

Case 4. :

- Outermost contraction is $(\mathbf{link'})$. Only non-trivial cases are where the innermost contraction is
  - $(\mathbf{subst'})\!:i, \mathbf{P}_{\bar{C} \rightsquigarrow x_j^r}$ on one of the linked modules:

Case $i \in \mathcal{F}$:

$$\begin{tikzcd}
& \bar{C}[\{\bar{D}\} \oplus^0 \{\bar{D}'\}] \arrow[dl, dashed, "(\mathbf{subst}')\,:\,i,\, \mathbf{P}_{\bar{C}\rightsquigarrow x_j^r}"'] \arrow[dr, dashed, "(\mathbf{link}')"] & \\
\bar{C}[\{\bar{D}[i := \bar{M}]\} \oplus^0 \{\bar{D}'\}] \arrow[dr, dashed, "(\mathbf{link}')"'] & & \bar{C}[\{\bar{D}[-\mathcal{F}], \bar{D}'[-\mathcal{F}], \bar{D}''\}] \arrow[dl, dashed, "(\mathbf{subst}')\,:\,i',\, \mathbf{P}_{\bar{C}\rightsquigarrow x_{j'}^r}"] \\
& \bar{C}[\{\bar{D}[-\mathcal{F}], \bar{D}'[-\mathcal{F}], \bar{D}''[i' := \bar{M}]\}] &
\end{tikzcd}$$

Case $i \notin \mathcal{F}$:

$$\begin{tikzcd}
& \bar{C}[\{\bar{D}\} \oplus^0 \{\bar{D}'\}] \arrow[dl, dashed, "(\mathbf{subst}')\,:\,i,\, \mathbf{P}_{\bar{C}\rightsquigarrow x_j^r}"'] \arrow[dr, dashed, "(\mathbf{link}')"] & \\
\bar{C}[\{\bar{D}[i := \bar{M}]\} \oplus^0 \{\bar{D}'\}] \arrow[dr, dashed, "(\mathbf{link}')"'] & & \bar{C}[\{\bar{D}[-\mathcal{F}], \bar{D}'[-\mathcal{F}], \bar{D}''\}] \arrow[dl, dashed, "(\mathbf{subst}')\,:\,i,\, \mathbf{P}_{\bar{C}\rightsquigarrow x_j^r}"] \\
& \bar{C}[\{\bar{D}[-\mathcal{F}][i' := \bar{M}], \bar{D}'[-\mathcal{F}], \bar{D}''\}] &
\end{tikzcd}$$

– $(\mathbf{gc\text{-}module}')\!:J$ on one of the linked modules:

$$\begin{tikzcd}
& \bar{C}[\{\bar{D}_1, \bar{D}_2\} \oplus^0 \{\bar{D}_1'\}] \arrow[dl, dashed, "(\mathbf{gc\text{-}module}')\,:\,J"'] \arrow[dr, dashed, "(\mathbf{link}')"] & \\
\bar{C}[\{\bar{D}_1\} \oplus^0 \{\bar{D}_1'\}] \arrow[dr, dashed, "(\mathbf{link}')"'] & & \bar{C}[\{\bar{D}_1[-\mathcal{F}], \bar{D}_2[-\mathcal{F}], \bar{D}_1'[-\mathcal{F}], \bar{D}_1''\}] \arrow[dl, dashed, "(\mathbf{gc\text{-}module}')\,:\,J"] \\
& \bar{C}[\{\bar{D}_1[-\mathcal{F}], \bar{D}_1'[-\mathcal{F}], \bar{D}_1''\}] &
\end{tikzcd}$$

No component in $\{\bar{D}_1, \bar{D}_2\}$ can "be in" both J and $\mathcal{F}$ as these are required to be unnamed or named proper, respectively.

- Outermost contraction is $(\mathbf{subst\text{-}letrec}')\!:\mathbf{P}_{\bar{C}\rightsquigarrow x_j^r}$. We consider the following possibilities for the location and behaviour of the innermost contraction:

Case "in $\mathrm{Body}(\bar D[j]) = \bar M$**"** :

$$\begin{tikzcd}
& \bar C'[\langle \bar C[x_j^r] \mid \bar D\rangle^R] \arrow[dl, dashed, "(\mathbf{subst\text{-}letrec}')\colon \mathbf{P}_{\bar C \rightsquigarrow x^r}"'] \arrow[dr, dashed, "\mathrm{cn}"] & \\
\bar C'[\langle \bar C[\bar M] \mid \bar D\rangle^R] \arrow[d, dashed, "\mathrm{cn}"] & & \bar C'[\langle \bar C[x_j^r] \mid \bar D[j := (F_j \triangleright x_j = \bar M')^{R'}]\rangle^R] \arrow[ddl, dashed, "(\mathbf{subst\text{-}letrec}')\colon \mathbf{P}_{\bar C \rightsquigarrow x^r}"] \\
\bar C'[\langle \bar C[\bar M'] \mid \bar D\rangle^R] \arrow[dr, dashed, "\mathrm{cn}"] & & \\
& \bar C'[\langle \bar C[\bar M'] \mid \bar D[j := (F_j \triangleright x_j = \bar M')^{R'}]\rangle^R] &
\end{tikzcd}$$

The middle-left leg is valid by the contextual closure of $\dashrightarrow'$.

Case "garbage collecting hole in $\bar C$**"** : (with the obvious abuse of notation)

$$\begin{tikzcd}
& \bar C'''[\langle \bar C'[(F_k \triangleright x_k = \bar C''[x_j^r])^0] \mid \bar D\rangle^R] \arrow[dl, dashed, "(\mathbf{subst\text{-}letrec}')\colon \mathbf{P}_{\bar C \rightsquigarrow x_j^r}"'] \arrow[dr, dashed, "\mathrm{cn}"] & \\
\bar C'''[\langle \bar C'[(F_k \triangleright x_k = \bar C''[\bar M])^0] \mid \bar D\rangle^R] \arrow[rr, dashed, "\mathrm{cn}"'] & & \bar C'''[\langle \bar M' \mid \bar D\rangle^R]
\end{tikzcd}$$

Case "duplicating hole in $\bar C$**"** : (with the obvious abuse of notation)

$$\begin{tikzcd}
& \bar C'[\langle \bar C[x_j^r] \mid \bar D\rangle^R] \arrow[dl, dashed, "(\mathbf{subst\text{-}letrec}')\colon \mathbf{P}_{\bar C \rightsquigarrow x_j^r}"'] \arrow[drr, dashed, "\mathrm{cn}"] & & \\
\bar C'[\langle \bar C[\bar M] \mid \bar D\rangle^R] \arrow[ddr, dashed, "\mathrm{cn}"] & & & \bar C'[\langle \bar C''[x_j^r, x_j^r] \mid \bar D\rangle^R] \arrow[dl, dashed, "(\mathbf{subst\text{-}letrec}')\colon \mathbf{P}_{\bar C \rightsquigarrow x_j^r}"] \\
& & \bar C'[\langle \bar C''[\bar M, x_j^r] \mid \bar D\rangle^R] \arrow[dl, dashed, "(\mathbf{subst\text{-}letrec}')\colon \mathbf{P}_{\bar C \rightsquigarrow x_j^r}"] & \\
& \bar C'[\langle \bar C''[\bar M, \bar M] \mid \bar D\rangle^R] & &
\end{tikzcd}$$

Case otherwise : Trivial.

- Outermost contraction is (**select**$'$). We cannot garbage collect the selected component (and thus leave the selection stuck) as garbage collection only can remove unnamed components.