A Fresh Calculus for Name Management

D.Ancona, E.Moggi

DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy
email: {davide,moggi}@disi.unige.it

Abstract. We define a basic calculus for name management, which is obtained by an appropriate combination of three ingredients: extensible records (in a simplified form), names (as in FreshML), computational types (to allow computational effects, including generation of fresh names). The calculus supports the use of symbolic names for programming in-the-large, e.g. it subsumes Ancona and Zucca's calculus for module systems, and for meta-programming (but not the intensional analysis of object level terms supported by FreshML), e.g. it subsumes (and improves) Nanevski and Pfenning's calculus for meta-programming with names and necessity. Moreover, it models some aspects of Java's class loaders.

1 Introduction

We introduce a basic calculus, called MML_ν^N, providing name management abilities, like those needed for programming in-the-large [Car97,AZ02]. In MML_ν^N names play the same role as in CMS [AZ02] (and in record calculi): they are used to name components of a module and to refer to external components which need to be provided from the outside (by a name resolver).

In CMS (and record calculi) names are taken from some infinite set N. In MML_ν^N the idea is to move from ordinary set theory to Fraenkel and Mostowski's set theory, where there is an alternative choice for N, namely the *FM-set* of atoms (which is potentially infinite). By taking N as the FM-set of atoms, we can have, as in FreshML [SPG03], a construct that generates a fresh name. In FreshML names are terms (and there is a type of names), so generation of a fresh name is denoted by $\nu x.e$, where x is a term variable which gets bound to the fresh name, and e is the term where the fresh name can be used. In MML_ν^N names occur both in types and in terms, and using x in place of a name X would entail a type system with dependent types (which would be problematic), thus we must use a different binder $\nu X.e$ for names.

To understand the type system and operational semantics of MML_ν^N (and FreshML) there is no need to be acquainted with FM-sets [GP99]. However, some key mathematical properties of FM-sets, like *equivariance* (i.e. invariance w.r.t. name permutations), are manifest in the type system and the operational semantics. Besides names $X \in \mathsf{N}$, the calculus has

- terms $e \in \mathsf{E}$, a closed term corresponds to an *executable* program;
- name resolvers, mapping names to terms, more specifically they denote partial functions $\mathsf{N} \xrightarrow{fin} \mathsf{E}$ with finite domain. We write $r.X$ for the term obtained by applying r to *resolve* name X.

Terms include fragments $b(r)e$, i.e. terms abstracted w.r.t. a resolver r, which denote functions $(\mathsf{N} \xrightarrow{fin} \mathsf{E}) \to \mathsf{E}$. We write $e\langle r\rangle$ for the term obtained by *linking* fragment e using resolver r.

Remark 1. If resolvers were included in terms, we would get a λ-calculus with extensible records [CM94], indeed a record amounts to a partial function mapping names (of components) to their values. More precisely, $b(r)e$ would become an abstraction $\lambda r.e$ and $e\langle r\rangle$ an application $e\ r$. We would also gain in expressivity. The main reasons for considering resolvers as second class terms, is to have a simpler type system (no need of subtyping), and to show that the embedding of CMS (and $\nu^\square$) is possible under very limited assumptions about resolvers.

The ability to generate a fresh name is essential to prevent accidental overriding of a resolver. If we know in advance what names need to be resolved within a fragment (we call such a fragment closed), then we can statically choose a name which is fresh (for that fragment). However, generic functions manipulating *open* fragments will have to generate fresh names at run-time. There are several reasons for working with open fragments: increase reusability, reduce the need for naming conventions (between independent developers), delay decisions.

MML_ν^N is able to express also run-time code generation (as formalized in [DP01,Nan02]), partial evaluation [JGS93,Dav96] and staging [Tah99,She01] (but formal embedding results appear difficult to establish). On the other hand, the calculus does not support the manipulation of object language syntax modulo α-conversion typical of FreshML, which relies on atom abstraction and concretion[1].

We present MML_ν^N as a monadic metalanguage, i.e. its type system makes explicit which terms have computational effects. Its operational semantics is given according to the general pattern proposed in [MF03], namely we give local simplification rules applicable non-deterministically (because semantic preserving), and computation steps executed in a deterministic order (because they may have computational effects). Generation of fresh names is a computational effect (this is the case also in the current version of FreshML[2]), thus typing $\nu X.e$ requires computational types. However, we consider other monadic operations (for imperative computations), for two reasons:

- to show that generation of fresh names co-exists smoothly with other computational effects;
- to allow arbitrary interleaving of software assembly activities (such as linking) and normal computational activities.

The calculus MML_ν^N subsumes the name management features of CMS and $\nu^\Box$, while overcoming some deficiencies and (unnecessary) complexities. We mention briefly the main differences with these two calculi (for the benefit of those already familiar with them).

- In CMS there is an infinite set of names (but a program uses only finitely many of them). CMS is a pure calculus (without computational effects), thus we could restrict to the fragment of MML_ν^N without computational types (and monadic operations), called ML^N.
  In CMS recursion is bundled in mixin, and removing it results in a very inexpressive calculus. On the contrary, ML^N is an interesting calculus even without recursion, and one can add recursion to it following standard approaches (in Section 3 we add mutual recursive declarations *let* ρ *in* e), which are *orthogonal* to the name management facilities.
- In $\nu^\Box$ the typing rules for $\Box$-types (related to those for necessity of S4 modal logic) are quite restrictive. Without these restrictions substitution would be unsound in the type system of $\nu^\Box$. In fact, when forming a fragment in $\nu^\Box$ all names occurring in the body are implicitly abstracted. Such restrictions have no reason to exist in MML_ν^N, because we allow multiple name resolvers, and fragments $b(r)e$ are formed by abstracting over one name resolver. Furthermore, making name resolvers explicit, avoid the need to introduce *non-standard* forms of substitution. In $\nu^\Box$ types are assigned to names at name generation time, while MML_ν^N follows the approach of mainstream module languages (that don't have generation of fresh names), i.e. different modules can assign to the same name different types (and values). Therefore, programming in $\nu^\Box$ forces an overuse of name generation, because the language restricts name reuse.

Finally, some aspects of Java multiple loaders [LY99] can be encoded naturally in MML_ν^N. More precisely, loaders are modeled by resolvers, whereas class loading is encoded by using the link construct of MML_ν^N. In this way, it is possible to mimic situations, where a class file can be loaded

[1] The main obstacle to include all name management abilities of FreshML is a language design issue, namely how to avoid dependent types.

[2] A previous version of FreshML [PG00] uses a more elaborate type system, which is able to mask the computational effects due to generation of fresh names.

several times by different user-defined loaders, and the same symbolic reference in the class file can be resolved in different ways throughout the execution of the program.

Summary. Section 2 presents syntax, type system and operational semantics of MML_ν^N, a monadic metalanguage for name management (and imperative computations). Section 3 introduces ML_Σ^N, a sub-extension of MML_ν^N with records and recursive definitions, and shows that in it one recovers in a natural way all (mixin) module operations of CMS. Section 4 gives several programming examples: programming with open fragments, benchmark examples for comparison with other calculi, and sample encoding of Java class loaders. Finally, Section 5 discusses related calculi.

Notation. In the paper we use the following notations and conventions.

- m range over the set $\mathcal{N}$ of natural numbers. Furthermore, $m \in \mathcal{N}$ is identified with the set $\{i \in \mathcal{N} | i < m\}$ of its predecessors.
- Term equivalence, written $\equiv$, is α-conversion. $\mathrm{FV}(e)$ is the set of variables free in e, while $e[x_i : e_i \mid i \in m]$ denotes parallel capture avoiding substitution.
- $f : A \xrightarrow{fin} B$ means that f is a partial function from A to B with a finite domain, written $\mathsf{dom}(f)$. $A \to B$ denotes the set of total functions from A to B. We use the following operations on partial (and total) functions:
  - $\{a_i : b_i | i \in m\}$ is the partial function mapping a_i to b_i (where the a_i must be different, i.e. $a_i = a_j$ implies $i = j$); in particular $\emptyset$ is the everywhere undefined partial function;
  - $f_{\setminus a}$ denotes the partial function f' defined as follows: $f'(a') = b$ iff $b = f(a')$ and $a' \neq a$;
  - $f\{a : b\}$ denotes the (partial) function f' s.t. $f'(a) = b$ and $f'(a') = f(a')$ otherwise;
  - f, f' denotes the union of two partial functions with disjoint domains.
- $X \# X'$ means that the sets X and X' are disjoint.

2 A basic calculus with names: MML_ν^N

This section introduces a monadic metalanguage MML_ν^N with *names*. Names X are syntactically pervasive, i.e. they occur both in types and in terms. Moreover, the term $\nu X.e$ allows to generate a *fresh* name for private use within e. Following FreshML [SPG03], we consider generation of a fresh name a computational effect, therefore for typing $\nu X.e$ we need computational types. In order to investigate the interactions of names with other computational effects, the metalanguage supports also imperative computations.

We parameterize typing judgements w.r.t. a finite set of names, namely those that can occur (free) in the judgement. The theoretical underpinning for manipulating names is provided by [GP99]. In particular, names can be permuted (but not unified), which suffices to consider terms up to α-conversion of bound names.

The operational semantics is given according to the general pattern proposed in [MF03], namely Section 2.3 specifies a confluent *simplification* relation $\longrightarrow$ (defined as the *compatible closure* of a set of rewrite rules), and Section 2.4 specifies a *computation* relation $\longmapsto$ describing how *configurations* may evolve. Most of the technical properties of MML_ν^N (and their proofs) are similar to those given in [MF03] for MML. Therefore, we shall skip most of the proof details.

The syntax is abstracted over symbolic names $X \in \mathsf{N}$, basic types b, locations $l \in \mathsf{L}$, term variables $x \in \mathsf{X}$ and resolver variables $r \in \mathsf{R}$. The syntactic category of types and signatures (i.e. the types of resolvers) is parameterized w.r.t. a finite set $\mathcal{X} \subseteq_{fin} \mathsf{N}$ of names that can occur in the types and signatures.

- $\boxed{\tau \in \mathsf{T}_\mathcal{X} ::= b \mid \tau_1 \to \tau_2 \mid [\Sigma|\tau] \mid M\tau \mid R\tau}$ $\mathcal{X}$-types, where
  $\Sigma \in \mathsf{\Sigma}_\mathcal{X} \stackrel{\Delta}{=} \mathcal{X} \xrightarrow{fin} \mathsf{T}_\mathcal{X}$ is a $\mathcal{X}$-signature $\{X_i : \tau_i | i \in m\}$

– $\boxed{\begin{array}{l} e \in \mathsf{E} ::= x \mid \lambda x.e \mid e_1\ e_2 \mid \theta.X \mid e\langle\theta\rangle \mid b(r)e \mid \\ \qquad\quad ret\ e \mid do\ x \leftarrow e_1; e_2 \mid \nu X.e \mid \\ \qquad\quad l \mid get\ e \mid set\ e_1\ e_2 \mid ref\ e \end{array}}$ terms, where
$\theta \in \mathsf{ER} ::= r \mid ? \mid \theta\{X : e\}$ is a name resolver term.

We give an informal semantics of the language (and refer to Section 4 for examples).

– The type $[\Sigma|\tau]$ classifies fragments which produce a term of type τ when linked with a resolver for Σ. The terms $\theta.X$ and $e\langle\theta\rangle$ use θ to *resolve* name X and to *link* fragment e. The term $b(r)e$ *represents* the fragment obtained by abstracting e w.r.t. r.
– The resolver ? cannot resolve any name, while $\theta\{X : e\}$ resolves X with e and *delegates* the resolution of other names to θ.
– The monadic type $M\tau$ classifies programs computing values of type τ. The terms *ret* e and $do\ x \leftarrow e_1; e_2$ are used to terminate and sequence computations, $\nu X.e$ generates a *fresh* name for use within the computation e.
– The reference type $R\tau$ classifies locations with values of type τ (locations l are instrumental to the operational semantics). The following monadic operations act on locations: *get* e returns the contents of a location, *set* $e_1\ e_2$ updates the contents of location e_1 with e_2, and *ref* e generates a new location with e as initial value.

As a simple example, let us consider the fragment `b(r)(r.X*r.X)` which can be correctly linked by resolvers mapping `X` to integer expressions and whose type is `[X:int|int]`. Then we can link the fragment with the resolver `?{X:2}`, as in `b(r)(r.X*r.X)<?{X:2}>`, and obtain `2*2` of type `int`. Note that `b(r)(r.X*r.X)` is not equivalent to `b(r)(r.Y*r.Y)`, whose type is `[Y:int|int]`. This is in clear contrast with what happens with variables and λ-abstractions: `\x->x*x` and `\y->y*y` are equivalent and have the same type. The sequel of this section is devoted to the formal definition of MML_ν^N. More interesting examples (with informal explanatory text) can be found in Section 4.

One can define (by induction on τ, e and θ) the following syntactic functions:

– the set $\mathrm{FV}(\_) \subseteq_{fin} \mathsf{N} \uplus \mathsf{X} \uplus \mathsf{R}$ of free names and variables in $\_$, in particular $\mathrm{FV}(\{X_i : \tau_i | i \in m\}) = (\cup_{i\in m}\mathrm{FV}(\tau_i)) \cup \{X_i | i \in m\}$
– the capture-avoiding substitution $\_[x_0 : e_0]$ for term variable x_0.
– the capture-avoiding substitution $\_[r_0 : \theta_0]$ for resolver variable r_0.
– the action $\_[\pi]$ of a name permutation π (with finite *support*) on $\_$.

2.1 Type system

The typing judgments are $\mathcal{X}; \Pi; \Gamma \vdash_\Omega e : \tau$ (i.e. e has type τ) and $\mathcal{X}; \Pi; \Gamma \vdash_\Omega \theta : \Sigma$ (i.e. θ resolves the names in the domain of Σ, and only them, with terms of the assigned type), where

– τ is a $\mathcal{X}$-type and Σ is a $\mathcal{X}$-signature
– $\Pi : \mathsf{R} \stackrel{fin}{\to} \Sigma_\mathcal{X}$ is a $\mathcal{X}$-signature assignment $\{r_i : \Sigma_i | i \in m\}$ for resolver variables
– $\Gamma : \mathsf{X} \stackrel{fin}{\to} \mathsf{T}_\mathcal{X}$ is a $\mathcal{X}$-type assignment $\{x_i : \tau_i | i \in m\}$ for term variables
– $\Omega : \mathsf{L} \stackrel{fin}{\to} \mathsf{T}_\mathcal{X}$ is an assignment of $\mathcal{X}$-types to locations.

The typing rules are given in Table 1. All the typing rules, except that for $\nu X.e$, use the same finite set $\mathcal{X}$ of names in the premises and the conclusion. The typing rule for $e\langle\theta\rangle$ supports a limited form of *width* subtyping, namely it allows to link a fragment $e : [\Sigma|\tau]$ with a resolver θ whose signature Σ' includes Σ. All the other rules are standard.

In the sequel we give the key properties of the type system needed for proving type safety. We write $\boxed{J ::= e : \tau \mid \theta : \Sigma}$, when it is unnecessary to distinguish between terms and name resolvers.

$$x\ \frac{\Gamma(x) = \tau}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega x : \tau} \qquad \text{lam}\ \frac{\mathcal{X}; \Pi; \Gamma, x : \tau_1 \vdash_\Omega e : \tau_2}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega \lambda x.e : \tau_1 \to \tau_2} \qquad \text{app}\ \frac{\mathcal{X}; \Pi; \Gamma \vdash_\Omega e_1 : \tau_1 \to \tau_2 \quad \mathcal{X}; \Pi; \Gamma \vdash_\Omega e_2 : \tau_1}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega e_1 e_2 : \tau_2}$$

$$\text{resolve}\ \frac{\mathcal{X}; \Pi; \Gamma \vdash_\Omega \theta : \Sigma \quad \tau = \Sigma(X)}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega \theta.X : \tau} \qquad \text{link}\ \frac{\mathcal{X}; \Pi; \Gamma \vdash_\Omega e : [\Sigma|\tau] \quad \mathcal{X}; \Pi; \Gamma \vdash_\Omega \theta : \Sigma'}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega e\langle\theta\rangle : \tau}\ \Sigma \subseteq \Sigma' \qquad \text{box}\ \frac{\mathcal{X}; \Pi, r : \Sigma; \Gamma \vdash_\Omega e : \tau}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega b(r)e : [\Sigma|\tau]}$$

$$r\ \frac{\Pi(r) = \Sigma}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega r : \Sigma} \qquad ?\ \frac{}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega ? : \emptyset} \qquad \text{extr}\ \frac{\mathcal{X}; \Pi; \Gamma \vdash_\Omega \theta : \Sigma \quad \mathcal{X}; \Pi; \Gamma \vdash_\Omega e : \tau}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega \theta\{X : e\} : \Sigma\{X : \tau\}}$$

$$\text{ret}\ \frac{\mathcal{X}; \Pi; \Gamma \vdash_\Omega e : \tau}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega ret\ e : M\tau} \qquad \text{do}\ \frac{\mathcal{X}; \Pi; \Gamma \vdash_\Omega e_1 : M\tau_1 \quad \mathcal{X}; \Pi; \Gamma, x : \tau_1 \vdash_\Omega e_2 : M\tau_2}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega do\ x \leftarrow e_1; e_2 : M\tau_2}$$

$$\nu\ \frac{\mathcal{X}, X; \Pi; \Gamma \vdash_\Omega e : M\tau}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega \nu X.e : M\tau}\ X \notin \mathrm{FV}(\Omega, \Pi, \Gamma, \tau) \qquad l\ \frac{\Omega(l) = \tau}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega l : R\tau} \qquad \text{get}\ \frac{\mathcal{X}; \Pi; \Gamma \vdash_\Omega e : R\tau}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega get\ e : M\tau}$$

$$\text{set}\ \frac{\mathcal{X}; \Pi; \Gamma \vdash_\Omega e_1 : R\tau \quad \mathcal{X}; \Pi; \Gamma \vdash_\Omega e_2 : \tau}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega set\ e_1\ e_2 : M(R\tau)} \qquad \text{new}\ \frac{\mathcal{X}; \Pi; \Gamma \vdash_\Omega e : \tau}{\mathcal{X}; \Pi; \Gamma \vdash_\Omega ref\ e : M(R\tau)}$$

Table 1. Type System for MML_ν^N

Lemma 1 (Equivariance). *If $\mathcal{X}; \Pi; \Gamma \vdash_\Omega J$, then $(\mathcal{X}; \Pi; \Gamma \vdash_\Omega J)[\pi]$, with π name permutation.*

Proof. By an easy induction on derivation of $\mathcal{X}; \Pi; \Gamma \vdash_\Omega J$.

Equivariance is a key property of names (the action of a name permutation is extended in the obvious way to typing judgments). The property cannot be improved, e.g. substitution of names with names fails to preserve typability. In fact, if we replace X_1 with X_2 in the signature $\{X_1 : \tau_1, X_2 : \tau_2\}$ (for simplicity assume $\mathrm{FV}(\tau_1, \tau_2) = \emptyset$) we get $\{X_2 : \tau_1, X_2 : \tau_2\}$, which is not a signature. However, if we swap X_1 and X_2 we get the signature $\{X_2 : \tau_1, X_1 : \tau_2\}$.

Lemma 2 (Weakening). *If $\mathcal{X} \subseteq \mathcal{X}'$ $\Pi \subseteq \Pi' : \mathsf{R} \xrightarrow{fin} \Sigma_{\mathcal{X}'}$, $\Gamma \subseteq \Gamma' : \mathsf{X} \xrightarrow{fin} \mathsf{T}_{\mathcal{X}'}$, $\Omega \subseteq \Omega' : \mathsf{L} \xrightarrow{fin} \mathsf{T}_{\mathcal{X}'}$ and $\mathcal{X}; \Pi; \Gamma \vdash_\Omega J$, then $\mathcal{X}'; \Pi'; \Gamma' \vdash_{\Omega'} J$*

Proof. By induction on derivation of $\mathcal{X}; \Pi; \Gamma \vdash_\Omega J$. In the case of binders the bound variable has to be renamed to avoid clashes with $\mathcal{X}'$, Π' and Γ'.

Weakening is a standard property of type systems. The statement is cumbersome, because we have to ensure that Π', Γ' and Ω' use only names in $\mathcal{X}'$.

Lemma 3 (Subsumption). *The following rules are admissible*

subsume-e *If $\mathcal{X}; \Pi, r : \Sigma_r; \Gamma \vdash_\Omega e : \tau$ and $\Sigma_r \subseteq \Sigma_r' \in \Sigma_{\mathcal{X}}$, then $\mathcal{X}; \Pi, r : \Sigma_r'; \Gamma \vdash_\Omega e : \tau$*
subsume-θ *If $\mathcal{X}; \Pi, r : \Sigma_r; \Gamma \vdash_\Omega \theta : \Sigma$ and $\Sigma_r \subseteq \Sigma_r' \in \Sigma_{\mathcal{X}}$, then $\mathcal{X}; \Pi, r : \Sigma_r'; \Gamma \vdash_\Omega \theta : \Sigma'$ for some $\Sigma \subseteq \Sigma' \in \Sigma_{\mathcal{X}}$*

Proof. By induction on derivation of $\mathcal{X}; \Pi, r : \Sigma; \Gamma \vdash_\Omega J$. The cases (resolve) and (link) are the only one that exploit directly the assumption $\Sigma_r \subseteq \Sigma_r'$.

Subsumption is peculiar of this type system, and is related to *width* subtyping.

Lemma 4 (Substitution). *The following rules are admissible*

$$\mathrm{Sub}_x\ \frac{\mathcal{X};\Pi;\Gamma \vdash_\Omega e_0 : \tau_0 \quad \mathcal{X};\Pi;\Gamma, x_0 : \tau_0 \vdash_\Omega J}{\mathcal{X};\Pi;\Gamma \vdash_\Omega e[x_0 : e_0] : \tau} \qquad \mathrm{Sub}_r\ \frac{\mathcal{X};\Pi;\Gamma \vdash_\Omega \theta_0 : \Sigma_0 \quad \mathcal{X};\Pi, r_0 : \Sigma_0;\Gamma \vdash_\Omega J}{\mathcal{X};\Pi;\Gamma \vdash_\Omega e[r_0 : \theta_0] : \tau}$$

Proof. By induction on the derivation of the second premises. Most cases are immediate or rely on the induction hypothesis (in combination with simple facts).

2.2 Polymorphic extension

Although the technical development will be restricted to the simply typed language, we sketch how to extend the type system with polymorphism, since it is essential for the example on open fragments generators (see Section 4). Basically we need to add type polymorphism (like that available in ML and Haskell) and row polymorphism [Rém93] (available in O'Caml). First we add type variables α and signature variables $p \in \mathsf{P}$. Then we extend the BNF for types and signatures and add the BNF for type schema:

- $\boxed{\tau \in \mathsf{T}_\mathcal{X} ::= b \mid \alpha \mid \tau_1 \to \tau_2 \mid [\Sigma|\tau] \mid M\tau \mid R\tau}$
- $\boxed{\Sigma \in \mathbf{\Sigma}_\mathcal{X} ::= \{X_i : \tau_i | i \in m\} \mid p, \{X_i : \tau_i | i \in m\}}$ where $X_i \in \mathcal{X}$ and $\tau_i \in \mathsf{T}_\mathcal{X}$ for any $i \in m$.
- $\boxed{\sigma \in \mathsf{S}_\mathcal{X} ::= \tau \mid \forall\alpha.\sigma \mid \forall p\#\mathcal{X}'.\sigma}$ where $\mathcal{X}' \subseteq \mathcal{X}$.

Intuitively, $p\#\mathcal{X}'$ means that p can be instantiated with a signature Σ provided $\mathsf{dom}(\Sigma)\#\mathcal{X}'$ (this is like the sorting of row variables in [Rém93]). The typing judgments $\mathcal{X};\Delta;\Pi;\Gamma \vdash_\Omega e : \tau$ have an additional component $\Delta : \mathsf{P} \to \mathcal{P}_{fin}(\mathcal{X})$, which assigns to every signature variable p its *sort*, moreover $\Gamma : \mathsf{X} \xrightarrow{fin} \mathsf{S}_\mathcal{X}$ assigns type schema instead of types. The typing rules of Table 1 are mostly unchanged, in the sense that Δ is the same in premises and conclusion, the only exceptions are

- the typing rule for a variable x, where one has to instantiate the type and signature variables quantified in $\Gamma(x)$. For instance $x\ \dfrac{\Gamma(x) = \forall p\#\mathcal{X}'.\tau}{\mathcal{X};\Delta;\Pi;\Gamma \vdash_\Omega x : \tau[p : \Sigma]}\ \mathcal{X};\Delta \vdash \Sigma\#\mathcal{X}'$
  where $\mathcal{X};\Delta \vdash \Sigma\#\mathcal{X}'$ means $\Sigma \in \mathbf{\Sigma}_\mathcal{X}$ and one of the following holds
  - $\Sigma \equiv \{X_i : \tau_i | i \in m\}$ and $\{X_i | i \in m\}\#\mathcal{X}'$, or
  - $\Sigma \equiv p, \{X_i : \tau_i | i \in m\}$ and $\{X_i | i \in m\}\#\mathcal{X}' \subseteq \Delta(p)$.

  The definition of $\_[p : \Sigma]$ is fairly straightforward.
- link $\dfrac{\mathcal{X};\Pi;\Gamma \vdash_\Omega e : [\Sigma|\tau] \quad \mathcal{X};\Pi;\Gamma \vdash_\Omega \theta : \Sigma'}{\mathcal{X};\Pi;\Gamma \vdash_\Omega e\langle\theta\rangle : \tau}\ \mathcal{X};\Delta \vdash \Sigma \subseteq \Sigma'$
  where $\mathcal{X};\Delta \vdash \Sigma \subseteq \Sigma'$ holds only in one of the following cases
  - $\Sigma \equiv \{X_i : \tau_i | i \in m\}$ and $\Sigma' \equiv \{X_i : \tau_i | i \in m + n\}$, or
  - $\Sigma \equiv \{X_i : \tau_i | i \in m\}$ and $\Sigma' \equiv p, \{X_i : \tau_i | i \in m + n\}$, or
  - $\Sigma \equiv p, \{X_i : \tau_i | i \in m\}$ and $\Sigma' \equiv p, \{X_i : \tau_i | i \in m + n\}$ and $X_{m+j} \in \Delta(p)$ for any $j \in n$.
- $\nu\ \dfrac{\mathcal{X}, X;\Delta^{+X};\Pi;\Gamma \vdash_\Omega e : M\tau}{\mathcal{X};\Delta;\Pi;\Gamma \vdash_\Omega \nu X.e : M\tau}\ X \notin \mathrm{FV}(\Omega, \Pi, \Gamma, \tau)$ where $\Delta^{+X}(p) = \Delta(p) \uplus \{X\}$.

Finally, we add the typing rule for let-binding. We give only an instance of it, to exemplify the effect on Δ let $\dfrac{\mathcal{X};\Delta\{p : \mathcal{X}'\};\Pi;\Gamma \vdash_\Omega e : \tau \quad \mathcal{X};\Delta;\Pi;\Gamma, x : \forall p\#\mathcal{X}'.\tau \vdash_\Omega e' : \tau'}{\mathcal{X};\Pi;\Gamma \vdash_\Omega let\ x = e\ in\ e' : \tau'}\ p \notin \mathrm{FV}(\Pi, \Gamma, \Omega)$.

2.3 Simplification

We define a confluent relation on terms, called *simplification*. There is no need to define a deterministic simplification strategy, since computational effects are *insensitive* to further simplification. Simplification $\longrightarrow$ is the compatible closure of the following rules

beta) $(\lambda x.e_2)\ e_1 \longrightarrow e_2[x : e_1]$
resolve) $(\theta\{X : e\}).X \longrightarrow e$
delegate) $(\theta\{X : e\}).X' \longrightarrow \theta.X'$ if $X' \neq X$
link) $(b(r)e)\langle\theta\rangle \longrightarrow e[r : \theta]$

Simplification enjoys the following properties.

Theorem 1 (CR). *The simplification relation* $\longrightarrow$ *is confluent.*

Theorem 2 (SR).

- *If* $\mathcal{X}; \Pi; \Gamma \vdash_\Omega e : \tau$ *and* $e \longrightarrow e'$*, then* $\mathcal{X}; \Pi; \Gamma \vdash_\Omega e' : \tau$*.*
- *If* $\mathcal{X}; \Pi; \Gamma \vdash_\Omega \theta : \Sigma$ *and* $\theta \longrightarrow \theta'$*, then* $\mathcal{X}; \Pi; \Gamma \vdash_\Omega \theta' : \Sigma$*.*

2.4 Computation

The computation relation $Id \longmapsto Id' \mid \mathsf{done}$ is defined using evaluation contexts, stores and configurations $Id \in \mathsf{Conf}$. A configuration records the current name space as a finite set $\mathcal{X}$ of names. The computation rules (see Table 2) consist of those given in [MF03] for the monadic metalanguage MML (these rules do not change the name space) plus generation of a fresh name (this is the only rule that extends the name space).

- $\boxed{E \in \mathsf{EC} ::= \Box \mid E[do\ x \leftarrow \Box; e]}$ evaluation contexts
- $\mu \in \mathsf{S} \stackrel{\Delta}{=} \mathsf{L} \stackrel{fin}{\rightharpoonup} \mathsf{E}$ stores map locations to their contents
- $(\mathcal{X}|\mu, e, E) \in \mathsf{Conf} \stackrel{\Delta}{=} \mathcal{P}_{fin}(\mathsf{N}) \times \mathsf{S} \times \mathsf{E} \times \mathsf{EC}$ configurations consist of the current name space $\mathcal{X}$ (which may grow as computation progresses), the current store μ, the program fragment e under consideration, and its evaluation context E
- $\boxed{rc \in \mathsf{RC} ::= ret\ e \mid do\ x \leftarrow e_1; e_2 \mid \nu X.e \mid get\ l \mid set\ l\ e \mid ref\ e}$ computational redexes.

Simplification $\longrightarrow$ is extended in the obvious way to a confluent relation on configurations (and related notions). The Bisimulation property, i.e. computation is insensitive to further simplification, is like that stated in [MF03] for MML.

Theorem 3 (Bisimulation). *If* $Id \equiv (\mathcal{X}|\mu, e, E)$ *with* $e \in \mathsf{RC}$ *and* $Id \stackrel{*}{\longrightarrow} Id'$*, then*

1. $Id \longmapsto D$ *implies* $\exists D'$ *s.t.* $Id' \longmapsto D'$ *and* $D \stackrel{*}{\longrightarrow} D'$
2. $Id' \longmapsto D'$ *implies* $\exists D$ *s.t.* $Id \longmapsto D$ *and* $D \stackrel{*}{\longrightarrow} D'$

where D *and* D' *range over* $\mathsf{Conf} \cup \{\mathsf{done}\}$*.*

One can also show that the simplification and computation relations are equivariant, i.e.

- if $Id \longrightarrow Id'$, then $Id[\pi] \longrightarrow Id'[\pi]$;
- if $Id \longmapsto D$, then $Id[\pi] \longmapsto D[\pi]$.

Administrative steps

(A.0) $(\mathcal{X}|\mu, ret\ e, \Box) \longmapsto \mathsf{done}$
(A.1) $(\mathcal{X}|\mu, do\ x \leftarrow e_1; e_2, E) \longmapsto (\mathcal{X}|\mu, e_1, E[do\ x \leftarrow \Box; e_2])$
(A.2) $(\mathcal{X}|\mu, ret\ e_1, E[do\ x \leftarrow \Box; e_2]) \longmapsto (\mathcal{X}|\mu, e_2[x : e_1], E)$

Name generation step

(ν) $(\mathcal{X}|\mu, \nu X.e, E) \longmapsto (\mathcal{X}, X|\mu, e, E)$ with X **renamed to avoid clashes**, i.e. $X \notin \mathcal{X}$

Imperative steps

(new) $(\mathcal{X}|\mu, ref\ e, E) \longmapsto (\mathcal{X}|\mu\{l : e\}, ret\ l, E)$ where $l \notin \mathsf{dom}(\mu)$
(get) $(\mathcal{X}|\mu, get\ l, E) \longmapsto (\mathcal{X}|\mu, ret\ e, E)$ with $e = \mu(l)$
(set) $(\mathcal{X}|\mu, set\ l\ e, E) \longmapsto (\mathcal{X}|\mu\{l = e\}, ret\ l, E)$ with $l \in \mathsf{dom}(\mu)$

Table 2. Computation Relation

$$\Box \quad \frac{}{\mathcal{X}; \Box : M\tau \vdash_\Omega \Box : M\tau} \qquad \frac{\mathcal{X}; \Box : M\tau_2 \vdash_\Omega E : M\tau' \quad \mathcal{X}; \emptyset; x : \tau_1 \vdash_\Omega e : M\tau_2}{\mathcal{X}; \Box : M\tau_1 \vdash_\Omega E[do\ x \leftarrow \Box; e] : M\tau'}$$

where $\mathcal{X}; \Box : M\tau \vdash_\Omega E : M\tau'$ is such that τ and τ' are $\mathcal{X}$-types and $\Omega : \mathsf{L} \overset{fin}{\to} \mathsf{T}_\mathcal{X}$.

Table 3. Well-formed Evaluation Contexts

2.5 Type Safety

Type safety, i.e. Subject Reduction and Progress properties, is like that established for MML in [MF03]. We expand only the case for $\nu X.e$, which relies on the equivariance property.

Definition 1 (Well-formed configuration). $\vdash_\Omega (\mathcal{X}|\mu, e, E) : \tau' \overset{\Delta}{\Longleftrightarrow}$

- *exists* $\tau \in \mathsf{T}_\mathcal{X}$ *s.t.* $\mathcal{X}; \emptyset; \emptyset \vdash_\Omega e : M\tau$ *and* $\mathcal{X}; \Box : M\tau \vdash_\Omega E : M\tau'$ *(see Table 3).*
- $\mathcal{X}; \emptyset; \emptyset \vdash_\Omega e_l : \tau_l$ *is derivable when* $e_l = \mu(l)$ *and* $\tau_l = \Omega(l)$

Lemma 5 (Equivariance). *If* $\mathcal{X}; \Box : M\tau \vdash_\Omega E : M\tau'$*, then* $(\mathcal{X}; \Box : M\tau \vdash_\Omega E : M\tau')[\pi]$.

Lemma 6 (Weakening). *If* $\mathcal{X} \subseteq \mathcal{X}'$*,* $\Omega \subseteq \Omega' : \mathsf{L} \overset{fin}{\to} \mathsf{T}_{\mathcal{X}'}$ *and* $\mathcal{X}; \Box : M\tau \vdash_\Omega E : M\tau'$*, then* $\mathcal{X}'; \Box : M\tau \vdash_{\Omega'} E : M\tau'$.

Theorem 4 (SR).

- *If* $\vdash_\Omega Id_1 : \tau'$ *and* $Id_1 \longrightarrow Id_2$*, then* $\vdash_\Omega Id_2 : \tau'$.
- *If* $\vdash_{\Omega_1} Id_1 : \tau'$ *and* $Id_1 \longmapsto Id_2$*, then exists* $\Omega_2 \supseteq \Omega_1$ *s.t.* $\vdash_{\Omega_2} Id_2 : \tau'$.

Proof. The second implication is proved by case analysis on the derivation of $Id_1 \longmapsto Id_2$. For the case (ν) we have $\vdash_{\Omega_1} Id_1 : \tau'$ and $Id_1 \equiv (\mathcal{X}|\mu, \nu X.e, E) \longmapsto (\mathcal{X}, X|\mu, e, E) \equiv Id_2$. Therefore, we should take $\Omega_2 = \Omega_1$ and derive $\vdash_{\Omega_2} Id_2 : \tau'$ by weakening (Lemma 2 and 6).

Lemma 7. *If* $\mathcal{X}; \emptyset; \emptyset \vdash_\Omega e' : \tau'$*, then* $e' \longrightarrow$ *or one of the following holds*

1. $\tau' \equiv \tau_1 \to \tau_2$ *and* $e' \equiv \lambda x.e$
2. $\tau' \equiv [\Sigma|\tau]$ *and* $e' \equiv b(r)e$

3. $\tau' \equiv R\tau$ *and* $e' \in \mathsf{L}$
4. $\tau' \equiv M\tau$ *and* $e' \in \mathsf{RC}$

If $\mathcal{X};\emptyset;\emptyset \vdash_\Omega \theta : \Sigma$, *then* $\mathsf{dom}(\Sigma) = \mathsf{dom}(\theta)$, *where* $\mathsf{dom}(\theta) \subseteq_{fin} \mathsf{N}$ *is defined by induction on* θ

$$\mathsf{dom}(r) \text{ is undefined} \qquad \mathsf{dom}(?) = \emptyset \qquad \mathsf{dom}(\theta\{X : e\}) = \mathsf{dom}(\theta) \cup \{X\}$$

Theorem 5 (Progress). *If* $\vdash_\Omega (\mathcal{X}|\mu, e, E) : \tau'$, *then*

1. *either* $e \notin \mathsf{RC}$ *and* $e \longrightarrow$
2. *or* $e \in \mathsf{RC}$ *and* $(\mathcal{X}|\mu, e, E) \longmapsto$

Proof. The second implication is proved by case analysis on $e \in \mathsf{RC}$. For the case $\nu X.e$ the proof is immediate, since the computation rule (ν) has no side-conditions.

3 Relating MML^N_ν with CMS

In this section we introduce ML^N_Σ a sub-extension of MML^N_ν. Then we define a translation of CMS in ML^N_Σ preserving CMS typing and reduction up to Ariola's equational axioms [AB02] for recursion.

The syntax of ML^N_Σ is defined in two steps. First, we remove from MML^N_ν computational and reference types (and consequently monadic operations, like $\nu X.e$, and locations). In the resulting calculus, called ML^N, the computation relation disappears (CMS is a pure calculus and its reduction semantics corresponds to simplification), the typing judgements are simplified $\mathcal{X};\Pi;\Gamma \vdash e : \tau$ (there is no need to have a type assignment to locations), and $\mathcal{X}$ could be left implicit, since the typing judgements of a derivation must use the same $\mathcal{X}$. Then we add records and mutual recursion:

- $\boxed{\tau \in \mathsf{T}_\mathcal{X} + = \Sigma}$ types, where $\Sigma \in \Sigma_\mathcal{X} \stackrel{\Delta}{=} \mathcal{X} \stackrel{fin}{\to} \mathsf{T}_\mathcal{X}$ is a $\mathcal{X}$-signature $\{X_i : \tau_i | i \in m\}$
- $\boxed{e \in \mathsf{E} + = o \mid e.X \mid e_1 + e_2 \mid e \setminus X \mid \mathit{let}\ \rho\ \mathit{in}\ e}$ terms, where

  $o : \mathsf{N} \stackrel{fin}{\to} \mathsf{E}$ is a record $\{X_i : e_i | i \in m\}$ and $\rho : \mathsf{X} \stackrel{fin}{\to} \mathsf{E}$ is a (recursive) binding $\{x_i : e_i | i \in m\}$.

The type $\Sigma \equiv \{X_i : \tau_i | i \in m\}$ classifies records of the form $\{X_i : e_i | i \in m\}$, i.e. with a fixed set of components. Notice that records should not be confused with resolvers. In particular, a fragment of type $[\Sigma|\tau]$ can be linked to a resolver of any signature $\Sigma' \supseteq \Sigma$. The operations on records correspond to the CMS primitives for mixins: $e.X$ selects the component named X, $e_1 + e_2$ concatenates two records (provided their component names are disjoint), and $e \setminus X$ removes the component named X (if present). The let construct allows mutually recursive declarations, which are needed for modeling the local components of a CMS module. The order of record components and mutually recursive declarations are immaterial, therefore o and ρ are not sequences (of binding) but functions (with finite domain).

Table 4 gives the typing rules for the new constructs. The properties of the type system in Section 2 extend in the obvious way to ML^N_Σ. We define simplification $\longrightarrow$ for ML^N_Σ as the compatible closure of the simplification rules for MML^N_ν (see Section 2.3) and the following simplification rules for record operations and mutually recursive declarations:

select) $o.X \longrightarrow e$ if $e \equiv o(X)$
plus) $o_1 + o_2 \longrightarrow o_1, o_2$ if $\mathsf{dom}(o_1)\#\mathsf{dom}(o_2)$
delete) $o \setminus X \longrightarrow o_{\setminus X}$
unfolding) $\mathit{let}\ \rho\ \mathit{in}\ e \longrightarrow e[x : \mathit{let}\ \rho\ \mathit{in}\ \rho(x) \mid x \in \mathsf{dom}(\rho)]$

Also simplification for ML^N_Σ enjoys confluence (Theorem 1) and subject reduction (Theorem 2).

$$o\ \frac{\{\mathcal{X};\Pi;\Gamma \vdash e_i : \tau_i \mid i \in m\}}{\mathcal{X};\Pi;\Gamma \vdash \{X_i : e_i | i \in m\} : \{X_i : \tau_i | i \in m\}} \qquad \text{select}\ \frac{\Sigma(X) = \tau \quad \mathcal{X};\Pi;\Gamma \vdash e : \Sigma}{\mathcal{X};\Pi;\Gamma \vdash e.X : \tau}$$

$$\text{plus}\ \frac{\mathcal{X};\Pi;\Gamma \vdash e_1 : \Sigma_1 \quad \mathcal{X};\Pi;\Gamma \vdash e_2 : \Sigma_2}{\mathcal{X};\Pi;\Gamma \vdash e_1 + e_2 : \Sigma_1, \Sigma_2}\ \mathsf{dom}(\Sigma_1)\#\mathsf{dom}(\Sigma_2) \qquad \text{delete}\ \frac{\mathcal{X};\Pi;\Gamma \vdash e : \Sigma}{\mathcal{X};\Pi;\Gamma \vdash e \setminus X : \Sigma \setminus X}$$

$$\text{rec}\ \frac{\{\mathcal{X};\Pi;\Gamma,\Gamma' \vdash \rho(x) : \Gamma'(x) \mid x \in \mathsf{dom}(\rho)\} \quad \mathcal{X};\Pi;\Gamma,\Gamma' \vdash e : \tau}{\mathcal{X};\Pi;\Gamma \vdash \mathit{let}\ \rho\ \mathit{in}\ e : \tau}\ \mathsf{dom}(\Gamma') = \mathsf{dom}(\rho)$$

Table 4. Additional Typing Rules for ML^N_Σ

CMS typing	ML^N_Σ typing
$\Gamma \vdash_{\mathsf{CMS}} E : \tau$	$\mathcal{X};\emptyset;\Gamma' \vdash E' : \tau'$
CMS type	ML^N_Σ type
$[\Sigma_1;\Sigma_2]$	$[\Sigma'_1 \mid \Sigma'_2]$
CMS term	ML^N_Σ term
x	x
$[\iota;o;\rho]$	$b(r)(\mathit{let}\ \rho'\ \mathit{in}\ o')[x : r.X \mid \iota(x) = X]$
$E_1 + E_2$	$b(r)E'_1\langle r\rangle + E'_2\langle r\rangle$
$E \setminus X$	$b(r)E'\langle r\rangle \setminus X$
$E.X$	$E'\langle ?\rangle.X$
$E!X$	$b(r)\mathit{let}\ \{x_1 : x_2.X, x_2 : E'\langle r\{X : x_1\}\rangle\}\ \mathit{in}\ x_2$
$C\{\rho\}$	$C'[\rho']$

the translations of Γ, Σ, o and ρ are defined pointwise.

Table 5. Translation of CMS in ML^N_Σ

3.1 Translation of CMS into ML^N_Σ

We refer to the Appendix (and [AZ99,AZ02]) for the definition of the CMS calculus. The key idea of the translation consists in translating a mixin type $[\Sigma_1;\Sigma_2]$ in $[\Sigma'_1|\Sigma'_2]$, in this way we obtain a compositional translation of CMS terms. In contrast, a translation based on functional types, where $[\Sigma_1;\Sigma_2]$ is translated in $\Sigma'_1 \to \Sigma'_2$, is not compositional (the problem is in the translation of $e_1 + e_2$, which must be driven by the type of e_1 and e_2).

Table 5 gives the translation of CMS in ML^N_Σ. Since CMS is parametric in the core language, the translation depends on the translation of core terms and types.

In the translation of a basic mixin $[\iota;o;\rho]$ the variables x in $\mathsf{dom}(\iota)$ (called *deferred*) are replaced with the resolution $r.X$ of the corresponding name $X = \iota(x)$, whereas the variables x in $\mathsf{dom}(\rho)$ (called *local*) are bound by the let construct for mutually recursive declarations. (A similar translation would not work in $\nu^\square$, because of the limitations in typing discussed in Section 5).

The translation of selection $E.X$ uses the empty resolver ?, since in CMS selection is allowed only for mixins without deferred components.

The freeze operator $E!X$ resolves a deferred component X with the corresponding output component. This resolution may introduce a recursive definition, since the output component X could be defined in terms of the corresponding deferred component. Therefore, the translation defines the record x_2 by resolving the name X with the X component of the record x_2 itself.

The typing preservation property of the translation can be proved easily, under the assumption that the property holds at the core level.

Theorem 6 (Typing preservation). *If $\Gamma \vdash_C C : c\tau$ implies $\emptyset; \Gamma' \vdash C' : c\tau'$ for every typing at the core level, then $\Gamma \vdash_{\mathsf{CMS}} E : \tau$ implies $\mathcal{X}; \emptyset; \Gamma' \vdash E' : \tau'$, where $\mathcal{X}$ includes all names occurring in the derivation of $\Gamma \vdash_{\mathsf{CMS}} E : \tau$.*

The translation preserves also the reduction semantics of CMS, but this can be proved only up to some equational axioms for mutually recursive declarations

$$\begin{array}{ll} \mathsf{C}[\mathit{let}\ \rho\ \mathit{in}\ e] = \mathit{let}\ \rho\ \mathit{in}\ \mathsf{C}[e] & \text{(lift)} \\ \mathit{let}\ \rho_1, x : (\mathit{let}\ \rho_2\ \mathit{in}\ e_2)\ \mathit{in}\ e_1 = \mathit{let}\ \rho_1, \rho_2, x : e_2\ \mathit{in}\ e_1 & \text{(merge)} \\ \mathit{let}\ \rho, x : e_1\ \mathit{in}\ e_2 = \mathit{let}\ \rho[x : e_1]\ \mathit{in}\ e_2[x : e_1] \text{ if } x \notin \mathrm{FV}(e_1) & \text{(sub)} \end{array}$$

The (lift) axiom corresponds to Ariola's lift axioms, in principle it can be instantiated with any ML^N_Σ context $\mathsf{C}[\ ]$, but for proving Theorem 7 it suffices to consider only the following contexts $\boxed{\mathsf{C}[\ ] ::= \Box + e \mid e + \Box \mid \Box \setminus X \mid \Box.X}$. The (merge) axiom is Ariola's internal merge, whereas (sub) is derivable from Ariola's axioms.

Let R denotes the set of the three axioms above, and S denotes the set of equational axioms corresponding to the simplification rules for ML^N_Σ; then the translation is proved to preserve the CMS reduction up to $=_{S \cup R}$ (i.e. the congruence induced by the axioms in $S \cup R$).

Theorem 7 (Semantics preservation). *If $E_1 \xrightarrow[\mathsf{CMS}]{} E_2$, then $E_1' =_{S \cup R} E_2'$.*

The translation of the non-recursive subset of CMS (i.e. no local declarations ρ and no freeze $E!X$) is a lot simpler, moreover its reductions are mapped to plain ML^N_Σ simplifications. However, non-recursive CMS is very in-expressive, e.g. one cannot translate the λ-calculus in it. On the contrary, we can define a translation $\_'$ of the λ-calculus in ML^N without functional types as shown below:

λ	ML^N without functional types
$\tau_1 \to \tau_2$	$[X : \tau_1' \mid \tau_2']$
x	x
$\lambda x.e$	$b(r)e'[x : r.X]$
$e_1\ e_2$	$e_1' \langle ?\{X : e_2'\} \rangle$

4 Programming examples

We demonstrate the use and expressivity of MML^N_ν with few paradigmatic examples:

- the first exemplifies programming with *open* fragments;
- the second and third are classical examples, to allow a comparison with other calculi for run-time code generation and staging;
- the forth exemplifies the analogies with Java class loaders.

To improve readability we use ML-like functional notation (note that in monadic metalanguages β-reduction is a sound simplification), ML-syntax for operations on references (`ref e`, `!e`, `e1:=e2`) and Haskell's do-notation `do {x1 <- e1; ...; xn <- en; e}`. In the sequence of commands of a do-expression we allow computations `ei` whose value is not bound to a variable (because it is not used by other commands) and non-recursive let-bindings like `xi = ei` (which amounts to replace `xi` with `ei` in the commands following the let-binding).

Example 1. We consider an example of generative programming, which motivates the use for fresh name generation. In our calculus a component can be identified with a fragment of type $[\Sigma|\tau]$, where Σ specifies what are the parameters that need to be provided for deployment. Generative programming support the dynamic manufacturing of customized components from elementary

(highly reusable) components. In our calculus the most appropriate building block for generative programming are polymorphic functions of type $G : \forall p.[p, \Sigma_i|\tau_i] \to M[p, \Sigma|\tau]$. The result type of G is computational, because generation may require computational activities, while the *signature variable* p classifies the information passed to the arguments of G, but not directly used or provided by G itself. Applications of G may instantiate p with different signatures, thus we say that G manipulates *open* fragments.

An over-simplified example of open fragment generator is

```
Ac: [p|a->a] -> M[p|{add: a -> M unit, update: M unit}]
```

it creates a data structure to maintain an (initially empty) set of accounts. Since we don't really need to know the structure of an account, we use a type variable `a`. The generator makes available two functionalities for operating on a set (of accounts): `add` inserts a new account in the set, and `update` modifies all the accounts in the set by applying a function of type `a->a`, which depends on certain parameters (e.g. interest rate) represented by the signature variable `p`. These parameters are decided by the bank after the data structure has been created, and they change over time. In many countries bank accounts are taxed, according to criteria set out by local authorities. So we need to provide a more refined generator

```
TaxedAc: [p'|a->a] -> [p|a->a] -> M[p'|[p|{add: a -> M unit, update: M unit}]]
```

the extra parameter computes the new balance based on the state of the account after the bank's update. We could define `TaxedAc` in terms of `Ac` as follows

```
fun TaxedAc tax upd = nu Tax.
    do {m <- Ac(b(r2) fn x => r2.Tax (upd<r2> x));
        ret (b(r') b(r1) m<r1{Tax:tax<r'>}>)};
```

Note that it is essential that the name `Tax` is fresh and private to `TaxedAc`, otherwise we may override some information in `r1`, which is needed by `upd`. In fact, `TaxedAc` is an open fragment generator that does not know in advance how the signature variable `p` could be instantiated. On the other hand, with *closed* fragment generators $G : [\Sigma_i|\tau_i] \to M[\Sigma|\tau]$ the problem does not arise, but reusability is impaired. For instance, it is not reasonable to expect that all banks will use the same parameters to update the accounts of their customers.

Example 2. We consider the classical power function `exp:int->real->M real`, which takes an exponent n and a base x, then it computes x^n by making recursive calls. Then we show how to get specialized versions (for fixed n) by unfolding the recursion at specialization time. The result type of `exp` is computational, because we consider recursion a computational effect.

```
(* standard power function *)
fun exp n x =  if n=0 then ret(1.0) else do {x' <- (exp (n-1) x); ret(x*x')};
>   exp = ... : int -> real -> M real
(* exp_c generates a fragment with hook X for base, its type reflects
   the fact that exp_c makes the recursive call at fragment generation time *)
fun exp_c n = if n=0 then ret(b(r) 1.0)
              else do {u <- exp_c (n-1); ret(b(r) (r.X * u<r>))};
>   exp_c = ... : int -> M[X:real | real]
(* optimized power function exp_o, its type is slightly different from
   that of exp, to reflects the different timing of recursive calls *)
fun exp_o n = do {u <- exp_c n; ret(fn x => u<?{X:x}>)};
>   exp_o = ... : int -> M(real -> real)
do  sq <- exp_o 2; (* performs the unfolding of recursive calls to exp_c *)
>   sq = (fn x => x*(x*1.0)) : real -> real
```

In comparison to [NP03, Example 3], we don't need fresh names and support polymorphism to give a simple definition of `exp_c` (in fact, we could avoid the use of names altogether). In comparison to MetaML [CMS03], we don't face the problems due to execution of *potentially open* code.

Example 3. We consider the imperative power function `p:int->real->(real ref)->M unit`, and show that we can recover the specialized version given in [CMS03, Section 4.1] without facing the problems due to *scope extrusion* (which were the main motivation for the introduction of closed types). The function `p` takes an exponent n, a base x and a reference `y`, then it initializes y with 1.0 and repeatedly multiplying the content of y with x until it becomes x^n. Therefore, the computational effects used by the imperative power function are recursion and side-effects.

```
(* imperative power function *)
fun p n x y =  if n=0 then y:=1.0 else do {p (n-1) x y; y' <- !y; y:=x*y'};
>   p = ... : int -> real -> (R real) -> M unit
(* p_c generates a fragment with hooks X and Y for base and location,
   its type reflects the fact that p_c makes the recursive call at
   fragment generation time, while the side-effects take place after
   the hooks have been resolved *)
fun p_c n = if n=0 then ret(b(r) r.Y:=1.0)
            else do {u <- p_c (n-1);
                     ret(b(r) do {u<r>; y' <-!r.Y; r.Y:=r.X*y'})};
>   p_c = ... : int -> M [X:real, Y:R real | M unit]
(* optimized imperative power function p_o, its type is slightly
   different from that of exp, to reflects the different timing of
   recursive calls *)
fun p_o n = do u <- p_c n in (fn x,y. u<?{X:x, Y:y}>);
>   p_o = ... : int -> M(real -> (R real) -> M unit)
do sq_i <- p_o 2; (* performs the unfolding of recursive calls to p_c *)
>   sq_i = (fn x,y => do {y:=1.0; y' <- !y; y:=x*y'; y' <- !y; y:=x*y'})
         : real -> (R real) -> M unit
```

Example 4. This example establishes a correspondence between our calculus and some key concepts behind Java class loaders [LY99]. Loaders are a powerful mechanism which allows dynamic linkage of code fragments, management of multiple name spaces and code instrumentation [LB98]. Some of the basic notions concerning Java multiple loaders can be encoded naturally in MML_ν^N, as suggested by the following table (we identify a class file f with a location containing a fragment):

Java	MML_ν^N
a loader	a resolver θ
loader delegation	$\theta\{X : e\}$
the content of a class file	$b(r)e$
a symbolic reference to a class X	$r.X$
loading of class file f with initiating loader θ	$do\ u \leftarrow get\ f; u\langle\theta\rangle$

As shown in the table, MML_ν^N resolvers play the role of class loaders which replace symbolic references with (concrete references [LY99] to) classes, and the ability to extend resolvers corresponds to a primitive form of delegation between loaders. As an example, consider the following code, where class files contain fragments of type `M int`, rather than class declarations.

```
(* create three class files *)
do f1 <- ref (b(r) return 1);
>  f1 = ... : R [ | M int]
```

```
do f2 <- ref (b(r) return 2);
>  f2 = ... : R [ | M int]
   f3 <- ref (b(r) do {x <- r.X; y <- r.Y; return (x+y)});
>  f3 = ... : R [X:M int, Y:M int | M int]
(* load class file f3 *)
do u  <- !f3;
>  u = ... : [X:M int, Y:M int | M int]
(* with initiating loader ?{X:c1, Y:c3} *)
do {c1 = do {u <- !f1; u<?>};
    c3 = do {u <- !f3;
             c2 = do {u <- !f2; u<?>};
             u<?{X:c2, Y:c2}>};
    u<?{X:c1, Y:c3}>};
>  5
```

There are three class files, `f1`, `f2` and `f3`; the execution of the program starts from class file `f3`. The *initiating loader* [LB98] for the main program is defined by `?{X:c1, Y:c3}`. According to the standard definition [LY99], the initiating loader of a given loaded class file f is the loader that will eventually try to resolve all symbolic references contained in f. Note that the same class file can be loaded more than once for resolving different symbolic references as happens in Java. For instance, in the program above the class file `f3` is loaded twice; the first time for starting execution of the main program, the second for resolving the symbolic reference `X` in the main program. Indeed, the two loaded code fragments are kept distinct and represent different entities as happens in Java when the same class is loaded by two different loaders. In this case the same symbolic reference in the same class file can be resolved in different ways. For instance, in the program above `Y` is resolved with `f1` in the main program and with `f2` when `f3` is loaded again.

A clear advantage of modeling Java loaders in MML_ν^N is a better support for code instrumentation (that is, the ability to change class bytecode at load-time), since in Java this feature is implemented at a very low level and basically consists in arbitrary and uncontrolled manipulation of bytecode. However not all aspects of Java loaders can be modeled in MML_ν^N, for instance there is no counterpart to dynamic typing.

5 Conclusions and related work

This section compares MML_ν^N with related calculi ($\nu^\Box$ and MMML).

The $\nu^\Box$ calculus of [Nan02,NP03] is a refinement of $\lambda^\Box$ [DP01], which provides better support for symbolic manipulation. The stated aim is to combine safely the best features of $\lambda^\Box$ (the ability to execute closed code) and $\lambda^\bigcirc$ [Dav96] (the ability to manipulate open code). The work on MetaML has similar aims, but adopt the opposite strategy, i.e. it starts from $\lambda^\bigcirc$. The monadic metalanguage MMML of [MF03] provides an operational semantics sufficiently detailed for analyzing subtle aspects of multi-stage programming [Tah99,She01,CMS03]), in particular the interactions between code generation and computational effects.

Comparison with $\nu^\Box$. The typing judgments of $\nu^\Box$ take the form $\Sigma;\Delta;\Gamma \vdash e : \tau[\mathcal{X}]$ where $\mathcal{X} \subseteq \mathsf{dom}(\Sigma)$ includes the names occurring *free* in e, and Δ has declarations of the form $u_i : \tau_i[\mathcal{X}_i]$ with $\mathcal{X}_i \subseteq \mathsf{dom}(\Sigma)$. Therefore, in $\nu^\Box$ the type of a name X is fixed globally (when the name is declared). This is a bad name space management policy, which goes against common practice in programming language design (e.g. modules and records).

In $\nu^\Box$ terms includes names, so our $\theta.X$ is replaced by X, in other words there is a *default resolver* which is left implicit. Linking $u\langle\Theta\rangle$ uses a function $\Theta \equiv \langle X_i \to e_i | i \in m\rangle$ to modify the

default resolver. The typing judgments for explicit substitutions take the form $\Sigma;\Delta;\Gamma \vdash \Theta : \mathcal{X}[\mathcal{X}']$, where $\mathcal{X}'$ includes the names *used* by the modified resolver to resolve the names in $\mathcal{X}$, e.g. $\mathcal{X} \subseteq \mathcal{X}'$ when Θ is empty. The following explicit substitution principle is admissible

$$\frac{\Sigma;\Delta;\Gamma \vdash \Theta : \mathcal{X}[\mathcal{X}'] \quad \Sigma;\Delta;\Gamma \vdash e : \tau[\mathcal{X}]}{\Sigma;\Delta;\Gamma \vdash e[\Theta] : \tau[\mathcal{X}']}$$

Our type $[\Sigma|\tau]$ is replaced by $\Box_{\mathcal{X}}\tau$, where $\mathcal{X} = \mathsf{dom}(\Sigma)$. The introduction rule for $\Box_{\mathcal{X}}\tau$ is

$$\frac{\Sigma;\Delta;\emptyset \vdash e : \tau[\mathcal{X}]}{\Sigma;\Delta;\Gamma \vdash box\ e : \Box_{\mathcal{X}}\tau[\mathcal{X}']}$$

This rule (inspired by modal logic) is very restrictive: it forbids having free term variables x in e, and acts like an *implicit binder* for the free names X of e (i.e. it binds the default resolver for e).

The observations above are formalized by a CBV translation $\_'$ of $\nu^{\Box}$-terms[3] into MML_{ν}^{N}, where the resolver variable r corresponds to the default resolver, which is implicit in $\nu^{\Box}$.

$e \in \nu^{\Box}$	$e' \in \mathsf{MML}_{\nu}^{N}$	$e \in \nu^{\Box}$	$e' \in \mathsf{MML}_{\nu}^{N}$
x	$ret\ x$	X	$r.X$
$\lambda x:\tau.e$	$ret\ (\lambda x.e')$	$u\langle X_i \to e_i\rangle$	$u\langle r\{X_i : e_i'\}\rangle$
$e_1\ e_2$	$do\ x_1 \leftarrow e_1'; x_2 \leftarrow e_2'; x_1 x_2$	$box\ e$	$ret\ (b(r)e')$
$\nu X:\tau.e$	$\nu X.e'$	$let\ box\ u = e_1\ in\ e_2$	$do\ u \leftarrow e_1'; e_2'$

We do not define the translation on types and assignments, since in $\nu^{\Box}$ the definition of well-formed signatures $\Sigma \vdash$ and types $\Sigma \vdash \tau$ is non-trivial.

In conclusion, the key novelty of MML_{ν}^{N} is to make name resolvers explicit and to allow a multiplicity of them, as a consequence we gain in simplicity and expressivity. Moreover, by building on top of a fairly simple form of extensible records, we are better placed to exploit existing programming language implementations (like O'Caml).

Comparison with MMML. We compare MMML and MML_{ν}^{N} at the level of the operational semantics. At the level of types, one expects an MMML code type $\langle\tau\rangle$ to correspond to a fragment type $[\Sigma|\tau]$, but it is unclear what signature Σ one should take.

In MML_{ν}^{N} linking $e\langle\theta\rangle$ and name resolution $\theta.X$ affect only the simplification relation. On the other hand in MMML code generation, e.g. $\lambda_M x.e$, affects the computation relation, i.e. it requires the generation of fresh names (moreover compilation may cause a run-time error, but this could be viewed as a weakness of the type system of MMML). In our calculus the computational effects due to code generation can be expressed as follows

- $\lambda_M x.e$ is a computation generating code for a λ-abstraction. In MML_{ν}^{N} it becomes

  $$\nu X.do\ u \leftarrow e[x : (b(r')r'.X)]; ret\ (b(r)\lambda x.u\langle r\{X : x\}\rangle)$$

  This term first computes a fragment u by evaluating e with x replaced by a fragment needing a resolver r' for the fresh name X (and possibly other names), then it returns a fragment for a λ-abstraction. Note that r does not have to resolve X, since u is linked to the modified resolver $r\{X : x\}$ (one also expects r' to be replaced by the modified resolver).
- $op_M(e)$, where op is a unary operation, is a computation generating code for a term of the form $op(\ldots)$, and does not generate fresh names. Thus in MML_{ν}^{N} it becomes $do\ u \leftarrow e; b(r)op(u\langle r\rangle)$. Note that in this case u is linked directly to r.

[3] In [NP03] the operational semantics (and the typing) of $\nu X.e$ differs from that adopted by (us and) FreshML. To avoid unnecessary complications, we work as if $\nu^{\Box}$ is FreshML compliant.

However, it is unclear how to device a whole translation from these two examples. Another feature of MMML (and MetaML) is cross-stage persistence $up(e)$, a.k.a. binary inclusion. The terms 0_V and $up(0)$ of code type $\langle \mathsf{int} \rangle$ are different, i.e. they cannot be simplified to a common term, and in *intentional analysis* one wants to distinguish them, since $up(e)$ is a black box for intentional analysis. In MML_ν^N it seems impossible to capture this difference. In conclusion, MML_ν^N might be as expressive as MMML, and its operational semantics appears to be at a lower level of detail.

References

[AB02] Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 117(1-3):95–178, 2002.

[AZ99] Davide Ancona and Elena Zucca. A primitive calculus for module systems. In *Proc. Int'l Conf. Principles & Practice Declarative Programming*, volume 1702 of *LNCS*, pages 62–79. Springer-Verlag, 1999.

[AZ02] D. Ancona and E. Zucca. A calculus of module systems. *J. Funct. Programming*, 12(2):91–132, March 2002. Extended version of [AZ99].

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pages 266–277, 1997.

[CM94] L. Cardelli and J. C. Mitchell. Operations on records. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 295–350. The MIT Press, Cambridge, MA, 1994.

[CMS03] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *J. Funct. Programming*, 13(3):545–571, 2003.

[Dav96] R. Davies. A temporal-logic approach to binding-time analysis. In *the Symposium on Logic in Computer Science (LICS '96)*, pages 184–195, New Brunswick, 1996. IEEE Computer Society Press.

[DP01] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[GP99] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *Proc. 14th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 214–224, July 1999.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[LB98] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*, volume 33(10) of *Sigplan Notices*, pages 36–44. ACM Press, October 1998.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Second edition, 1999.

[MF03] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. In *Proc. FoSSaCS '03*, volume 2620 of *LNCS*. Springer-Verlag, 2003.

[Nan02] Aleksandar Nanevski. Meta-programming with names and necessity. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP-02)*, ACM SIGPLAN notices, New York, October 2002. ACM Press.

[NP03] A. Nanevski and F. Pfenning. Meta-programming with names and necessity. Submitted, 2003.

[Ore] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`. Last viewed August 1999.

[PG00] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Proc. Mathematics of Program Construction, 5th Int'l Conf. (MPC 2000)*, volume 1837 of *LNCS*, pages 230–255, Ponte de Lima, Portugal, July 2000. Springer-Verlag.

[Rém93] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming: Types, Semantics and Language Design*. MIT Press, 1993.

[She01] T. Sheard. Accomplishments and research challenges in meta-programming. In W. Taha, editor, *Proc. of the Int. Work. on Semantics, Applications, and Implementations of Program Generation (SAIG)*, volume 2196 of *LNCS*, pages 2–46. Springer-Verlag, 2001.

[SPG03] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. Freshml: Programming with binders made simple. In *Proc. 8th Int'l Conf. Functional Programming*. ACM Press, 2003.

[Tah99] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [Ore].

A Definition of CMS

Terms $\boxed{\begin{array}{l} E \in \mathsf{CMSE} ::= x \mid C\{\rho\} \mid [\iota; o; \rho] \mid E_1 + E_2 \mid E \setminus X \mid E!X \mid E.X \\ \quad C \in \mathsf{CE} ::= x \mid \ \ldots \end{array}}$ where

$x \in \mathsf{X}$ and $X \in \mathsf{N}$ (as in ML^N), $\iota : \mathsf{X} \xrightarrow{fin} \mathsf{N}$, $o : \mathsf{N} \xrightarrow{fin} \mathsf{CMSE}$ and $\rho : \mathsf{X} \xrightarrow{fin} \mathsf{CMSE}$
Free variables are defined as follows (we omit the trivial cases):
$\mathrm{FV}(C\{\rho\}) = \mathrm{FV}(\rho)$, $\mathrm{FV}([\iota; o; \rho]) = (\mathrm{FV}(o) \cup \mathrm{FV}(\rho)) \setminus (\mathsf{dom}(\iota) \cup \mathsf{dom}(\rho))$.
We assume the following implicit conditions for well-formed terms:
$\mathrm{FV}(C) \subseteq \mathsf{dom}(\rho)$ in $C\{\rho\}$, and $\mathsf{dom}(\iota)\#\mathsf{dom}(\rho)$ in $[\iota; o; \rho]$.
Such conditions are omitted in typing rule (mixin) and reduction rules (sub) and (plus).

Types $\boxed{\tau \in \mathsf{CMST} ::= c\tau \mid [\Sigma_1; \Sigma_2]}$ where $\Sigma : \mathsf{N} \xrightarrow{fin} \mathsf{CMST}$

Typing judgments $\boxed{\begin{array}{l} \Gamma \vdash_{\mathsf{CMS}} E : \tau \ \text{(module level)} \\ \Gamma \vdash_{\mathsf{C}} C : c\tau \quad \text{(core level)} \end{array}}$ where $\Gamma : \mathsf{X} \xrightarrow{fin} \mathsf{CMST}$.

Typing rules

Two signatures Σ_1 and Σ_2 are *compatible* iff $\Sigma_1(X) = \Sigma_2(X)$ for all $X \in \mathsf{dom}(\Sigma_1) \cap \mathsf{dom}(\Sigma_2)$.

(var) $\dfrac{}{\Gamma \vdash_{\mathsf{CMS}} x : \tau}\ \Gamma(x) = \tau$ (delete) $\dfrac{\Gamma \vdash_{\mathsf{CMS}} e : [\Sigma_1; \Sigma_2]}{\Gamma \vdash_{\mathsf{CMS}} e \setminus X : [\Sigma_1; \Sigma_2 \setminus X]}$

(core) $\dfrac{\{\Gamma \vdash_{\mathsf{CMS}} \rho(x) : c\tau_x \mid x \in \mathsf{dom}(\rho)\} \quad \{x : c\tau_x \mid x \in \mathsf{dom}(\rho)\} \vdash_{\mathsf{C}} C : c\tau}{\Gamma \vdash_{\mathsf{CMS}} C\{\rho\} : c\tau}$

(mixin) $\dfrac{\{\Gamma, \Gamma_1, \Gamma_2 \vdash_{\mathsf{CMS}} o(X) : \Sigma_2(X) \mid X \in \mathsf{dom}(o)\} \quad \{\Gamma, \Gamma_1, \Gamma_2 \vdash_{\mathsf{CMS}} \rho(x) : \Gamma_2(x) \mid x \in \mathsf{dom}(\rho)\}}{\Gamma \vdash_{\mathsf{CMS}} [\iota; o; \rho] : [\Sigma_1; \Sigma_2]}$ $\begin{array}{l} \mathsf{dom}(\Gamma_2) = \mathsf{dom}(\rho) \\ \mathsf{dom}(\Sigma_1) = \mathsf{img}(\iota) \text{ and } \Gamma_1 = \Sigma_1 \circ \iota \\ \mathsf{dom}(\Sigma_2) = \mathsf{dom}(o) \end{array}$

(plus) $\dfrac{\Gamma \vdash_{\mathsf{CMS}} e_1 : [\Sigma_1^1; \Sigma_2^1] \quad \Gamma \vdash_{\mathsf{CMS}} e_2 : [\Sigma_1^2; \Sigma_2^2]}{\Gamma \vdash_{\mathsf{CMS}} e_1 + e_2 : [\Sigma_1^1, \Sigma_1^2; \Sigma_2^1, \Sigma_2^2]}$ $\begin{array}{l} \Sigma_1^1 \text{ compatible with } \Sigma_1^2 \\ \mathsf{dom}(\Sigma_2^1)\#\mathsf{dom}(\Sigma_2^2) \end{array}$

(freeze) $\dfrac{\Gamma \vdash_{\mathsf{CMS}} e : [\Sigma_1; \Sigma_2]}{\Gamma \vdash_{\mathsf{CMS}} e!X : [\Sigma_1 \setminus X; \Sigma_2]}\ \Sigma_1(X) = \Sigma_2(X)$ (select) $\dfrac{\Gamma \vdash_{\mathsf{CMS}} e : [\emptyset; \Sigma]}{\Gamma \vdash_{\mathsf{CMS}} e.X : \tau}\ \tau = \Sigma(X)$

Reduction rules

The one step reduction relation $\xrightarrow[\mathsf{CMS}]{}$ is closed under arbitrary contexts CMSC with one hole

(context) $\dfrac{E_1 \xrightarrow[\mathsf{CMS}]{} E_2}{\mathsf{CMSC}[E_1] \xrightarrow[\mathsf{CMS}]{} \mathsf{CMSC}[E_2]}$ (we omit to spell out the definition of such contexts)

(sub) $C_1\{\rho_1, x : C_2\{\rho_2\}\} \xrightarrow[\mathsf{CMS}]{} C_1[x : C_2]\{\rho_1, \rho_2\}$

(plus) $[\iota_1; o_1; \rho_1] + [\iota_2; o_2; \rho_2] \xrightarrow[\mathsf{CMS}]{} [\iota_1, \iota_2; o_1, o_2; \rho_1, \rho_2]$ if $\mathsf{dom}(o_1)\#\mathsf{dom}(o_2)$

(delete) $[\iota; o; \rho] \setminus X \xrightarrow[\mathsf{CMS}]{} [\iota; o_{\setminus X}; \rho]$

(freeze) $[\iota, \{x : X\}; o, \{X : E\}; \rho]!X \xrightarrow[\mathsf{CMS}]{} [\iota; o, \{X : E\}; \rho, \{x : E\}]$

(select) $[; o, \{X : E\}; \rho].X \xrightarrow[\mathsf{CMS}]{} E[x : [; X : \rho(x); \rho].X \mid x \in \mathsf{dom}(\rho)]$