Program Generation and Components

D.Ancona, E.Moggi[⋆]

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

Abstract. The first part of the paper gives a brief overview of meta-programming, in particular program generation, and its use in software development. The second part introduces a basic calculus, related to FreshML, that supports program generation (as described through examples and a translation of MetaML into it) and programming in-the-large (this is demonstrated by a translation of CMS into it).

1 Introduction

This paper explains what is program generation and what are the most promising uses of it, recalls the role of names in software components, and then presents a basic calculus, called MML^N_ν, which in the authors' view is a suitable formalism to describe program generation in terms of more primitive notions, namely name generation, name resolution and linking.

In calculi of module systems as CMS [AZ99,MT00,WV00,AZ02] modules can refer to *deferred* components by means of names. These calculi provide primitive operators for linking modules and resolving external names of deferred components, thus supporting programming in-the-large [Car97]. Analogously, in the MML^N_ν calculus of [AM04] names are used to refer to external components which need to be provided from the outside (by a name resolver).

In module (and record) calculi names are taken from some infinite set. On the contrary, in MML^N_ν at any time during execution there is a finite set of names, which can be extended dynamically by a construct $\nu X.e$ for generating a *fresh* name, borrowed from FreshML of [SPG03].

Fraenkel and Mostowski's set theory (see [GP99]) provides the mathematical underpinning of name generation for FreshML and MML^N_ν, but to understand the operational semantics and type system there is no need to be acquainted with FM-sets. Besides names $X \in \mathsf{Name}$, the calculus has

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

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

[⋆] Supported by EU projects DART IST-2001-33477 and APPSEM-II IST-2001-38957

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, $\mathsf{b}(r)e$ would become an abstraction $\lambda r.e$ and $e\langle r\rangle$ an application $e\ r$. Even when resolvers are second class terms, one can express in the calculus the staging constructs of MetaML [Tah99,She01] and the mixin operations of CMS.

The ability to generate a fresh name is essential to prevent accidental overriding. 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: reusability is increased, the need for naming conventions (between independent developers) is reduced, and decisions can be delayed.

We present MML^N_ν as a monadic metalanguage, i.e. its type system makes explicit which terms have computational effects, and its operational semantics is given by a (semantic preserving) simplification relation on terms and a computation relation on *configurations*. Generation of fresh names is a computational effect, as in FreshML, thus typing $\nu X.e$ requires computational types.

Summary. Section 2 explains program generation within the broader context of meta-programming, and mentions its most promising uses. Section 3 recalls the role of names in software components. Section 4 recalls syntax, type system and operational semantics of MML^N_ν. Section 5 gives several programming examples: programming with open fragments, and benchmark examples for comparison with other calculi (in particular MetaML). Section 6 introduces a 2-level version of MetaML, and show how it can be translated into MML^N_ν. Section 7 introduces ML^N_Σ, a variant of MML^N_ν with records and recursive definitions, where one can recover all (mixin) module operations of CMS. Finally, Section 8 discusses related calculi based on names, i.e. FreshML of [SPG03] and $\nu^\Box$ of [NPar].

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

- m ranges 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 $\equiv$ is α-conversion. $\mathrm{FV}(e)$ is the set of variables free in e, while $e[x_i\colon e_i \mid i \in m]$ denotes parallel capture avoiding substitution.
- $f\colon A \overset{fin}{\to} B$ means that f is a partial function from A to B with a finite domain, written $\mathsf{dom}(f)$. The image of f is denoted by $\mathsf{img}(f)$. $A \to B$ denotes the set of total functions from A to B. We use the following operations:
  - $\{a_i\colon b_i | i \in m\}$ is the partial function mapping a_i to b_i (where the a_i must be different, i.e. $a_i = a_j$ implies $i = j$);
    $\emptyset$ is the everywhere undefined partial function;
  - $f_{\backslash a}$ denotes the partial function f' s.t. $f'(a') = b$ iff $b = f(a')$ and $a' \neq a$;
  - $f\{a\colon b\}$ denotes the (partial) function f' s.t. $f'(a) = b$ and $f'(a') = f(a')$ when $a' \neq a$;
  - f, f' denotes the union of two partial functions with disjoint domains.
- $A\#B$ means that the sets A and B are disjoint.

2 Program generation

We explain what is program generation by placing it in the broader context of meta-programming. We borrow from Sheard's invited talk at SAIG'01 [She01], which in addition discusses several areas of meta-programming research. Then we mention some of the most promising uses of program generation in the context of software development. We make no attempt to be exhaustive, instead we advise the interested reader to browse through the proceedings of the conference on Generative Programming and Component Engineering (GPCE) [BCT02,PS03,KV04], and a compendium [LBCO04] of contributions presented at a Dagstuhl seminar on "Domain-specific program generation", where a new IFIP WG on "Program Generation" was proposed.

2.1 What is it?

In general programs manipulate data. Meta-programs are programs that manipulate object-programs, or more precisely data representing other programs. We are not committed to a particular (programming) language, thus by object-program we mean a syntactic element in a formal language. Broadly speaking we can classify meta-programs in three categories:

– Generators, which construct object-programs. For instance, specializers generate a specialized program solving an instance of a general problem.
– Analyzers, which analyze the structure of object-programs. For instance, type-checkers or tools that perform various kinds of static analysis.
– Transformers, which combine the features of analyzers and generators. For instance, optimizers that perform source-to-source transformation, or aspect weavers that insert code to address a cross-cutting concern.

A compiler is a typical example of program where the three kinds of meta-programs co-exist: a static analyzer for the source language, an optimizer for some intermediate language, and a program generator for the target language.

Object-programs can be represented at different level of abstraction (we call *code* the data representing a program):

– White-box abstraction, where code is text (i.e. a string) or an abstract syntax tree (AST). This representation is the most versatile, since it gives a low-level description of the object-program, but could be error prone.
– White-box abstraction modulo α-conversion. This representation of syntax has been considered in the context of logical frameworks to deal with binders in object languages, alongside other representations like higher-order abstract syntax. FreshML [GP99,SPG03] is the leading programming language that supports this abstraction.
– Black-box abstraction, where code can be executed and combined, but not analyzed. This is the most abstract representation.

The black-box abstraction is incompatible with program analyzers and transformers. On the other hand, program generators can work with any of the abstractions, and the black-box abstraction ensures the maximum separation of concerns between the generator and the user of the generated code.

We consider related concepts to clarify how they differ from the concept of code and meta-program. A program *configuration* is a snapshot of a program during execution, a *computation* is a description of the program execution, while code represents (the syntax of) a program independently from execution. A reflective program is a program that manipulates *itself*, thus it is a particular instance of a meta-program. One can identify three forms of reflection:

– Introspection is the ability of a program to analyze itself, namely its code (this introspection is called structural reflection) or its current configuration or execution history (this introspection is called behavioral reflection).
– Self-modification is the ability to modify itself.
– Intercession is the ability to manipulate its semantics, i.e. the interpreter or virtual machine for the reflective program.

A computation is *staged* when it is decomposed into stages along the temporal dimension. The change of stage is usually triggered by the acquisition of new information. In a meta-programming system supporting program generation there are two natural stages: the computation of the meta-program and the computation of the generated object-program. Depending on the nature of the meta-program, its computation is called differently (e.g. generation-time, compile-time, design-time, specialization-time), while the computation of the object-program is usually called run-time or use-time computation. Moreover, if the meta-programming system is *homogeneous*, i.e. the object-language for describing object-programs coincides with the meta-language, then it provides a natural support for *multi-stage* programming. MetaML [TS97] and MetaOCaml [CTHL03,Met01] are among the leading multi-stage programming languages. For several applications *heterogeneous* meta-programming systems are enough. In this case, the main issue is to provide support for a variety of object-languages.

2.2 What is for?

Generative and component approaches have the potential to revolutionize software development in a similar way as automation and components revolutionized manufacturing. Generative programming (developing programs that synthesize other programs), component engineering (raising the level of modularization and analysis in application design), and domain-specific languages (elevating program specifications to compact domain-specific notations that are easier to write and maintain) are key technologies for automating program development. Before focusing on program generation, we mention some trends in software engineering, in order to provide a broader picture. [GS04] identifies *software factories* as the next methodology for software development. A software factory is

> a collection of *reusable* assets (like patterns, models, frameworks, tools) for *rapidly* and *cheaply* producing an *open-ended* set of unique variants of a software *product*.

Clearly, for a software product that has a big market and need to evolve, like an operating system, this is an economically feasible approach. However, to make software factories economically feasible for specific domains, it is essential to empower the domain-experts and end-users.

Domain-specific languages (DSLs) are a way to give programming abilities to domain-experts and end-users. Descriptions given in a DSL can be treated as high-level source code, rather than non-executable requirement specifications. Relational databases are a "classic" example of success story in the use of DSLs, while UML is a counter-example of DSL. In fact, UML is too general (i.e. it is not meant for any specific domain) and too imprecise (e.g. it cannot be used as source code, although some subsets might). Other examples of domains where DSL technology has been used successfully or appear highly promising are: language parsers, reactive real-time programs and the telephony domain. For instance, MetaCase Consulting (http://www.metacase.com) gave a demo at GPCE'03 entitled "MetaEdit+ revolutionized the way Nokia develops mobile phone software", and the choice of MetaEdit+ was motivated as follows

> When Nokia was searching for an effective CASE tool, the prime criteria was encapsulation of domain knowledge, flexible method support and code generation. After evaluating a number of off-the-shelf CASE tools, they undertook the development of their own solution using the MetaEdit+ metaCASE tool.

DSLs provide language-based abstraction, which goes well beyond the abstraction provided by libraries (i.e. platform extension). In this context program generators play the role of compiler back-ends for domain-specific languages, and provide the following benefits

– Increase automation by exploiting domain features and knowledge.
– Improve performance via partial evaluation.

Usually program generators are *co-designed* with the DSL they implement, unlike compilers for general-purpose languages. It is worth to adopt this approach, when the effort to implement a program generator is comparable to that of implementing a software library for the specific domain. In the DSL approach there are other things one can do before the program generation stage, namely

– analysis, which exploits domain-knowledge to identify problems prior generation, or to provide static guarantees
– transformation, for instance to perform domain-specific optimization.

It is important that analysis and transformation take place before program generation, so that they can be understood and managed by the user of the DSL, who can be expected to be knowledgeable of the domain but not of the target language for the program generator.

3 Names and software components

It has been argued that the notion of software component is so general that cannot be defined in a precise and comprehensive way [CE00]. For instance, [Szy02] provides three different definitions, that adopt different levels of abstraction. Nevertheless, names play an important role, independently of the particular definition adopted for software components. For clarity, we identify the notion of software component with that of *mixin module*[1], as done in CMS of [AZ02].

A basic module could be described as follows

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

A basic module make use of *names* and *variables*. The former are the names of the components the module either *imports* from the outside (*input* components X_1, …, X_m) or *exports* to the outside (*output* components Y_1,…, Y_n). The latter are the variables used in definitions inside the module (i.e. in the expressions E_1,…, E_n, E'_1,…, E'_p). These variables can be either *deferred* (x_1,…, x_n), i.e. associated with some input component, or locally defined (z_1,…, z_p).

The distinction between names and variables is crucial: names correspond to external references, while variables correspond to internal references. Technically speaking the main differences between variables and names are: expressions include variables but not names; variables declared in a basic module are local and can be α-converted; while component names belong to a global name space and allow modules to talk to each other.

A useful operator which can be easily encoded in CMS is the *link* operator $\mathit{link}(M_1, M_2)$, used for merging two modules and resolving input names. This operator may be regarded as either an operation provided by a module language in order to define structured module expressions or an extra-linguistic mechanism to combine object files provided by a tool for modular software development. $\mathit{link}(M_1, M_2)$ is well-defined if the sets of the output components of M_1 and M_2 are disjoint. In this case, $\mathit{link}(M_1, M_2)$ corresponds to a module where some input component of one module has been bound to the definition of the corresponding output component of the other module, and conversely.

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

```
module BOOL is
  import IntEv as ext_ev
  export BoolEv = ev
  local
   fun ev EQ(ie1,ie2) = ext_ev(ie1)==ext_ev(ie2)
       |  ...
end BOOL;
```

[1] In the sequel we will interchangeably use the terms “module” and “mixin” as abbreviations of “mixin module”.

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

The result of *link*(BOOL,INT) corresponds to the module

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

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

The *link* operation described above can be decomposed in two steps. First, put together the declarations of the two arguments in one module, yielding

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

Then, bind import components with export components with the same name, yielding `BOOL_INT`. Formally, this corresponds to the fact that *link* is a derived operator which can be expressed by the *sum* and *freeze* basic operators of CMS.

CMS provides also a primitive operation for deleting module components, which allows redefinition of components when used in conjunction with the *link* operator. This is an important feature for enabling reuse of software components and amortizing the investment over multiple applications [Szy02].

4 A core calculus with names: MML_{ν}^{N}

This section recalls the monadic metalanguage MML_{ν}^{N} of [AM04]. For simplicity, we focus on the key feature, i.e. *names*, and exclude imperative computations and functional types. Moreover, we restrict the formal treatment to a simply typed language (Section 4.1), and recall only the main statements concerning type safety (Section 4.4). We refer to [AM04] for details and a polymorphic extension of the type system, which is essential for typing the examples on open fragments generators (see Example 1 in Section 5). The operational semantics is given according to the general pattern proposed in [MF03], namely by a confluent *simplification* relation $\Longrightarrow$ defined as the *compatible closure* of a set of rewrite rules (see Section 4.2), and a *computation* relation $\longmapsto$ describing how *configurations* may evolve (see Section 4.3).

Names X are syntactically pervasive, i.e. they occur both in types and in terms. The term $\nu X.e$ allows to generate a *fresh* name for private use within e. Following FreshML of [SPG03], we consider generation of a fresh name a computational effect, therefore for typing $\nu X.e$ we need computational types.

We parameterize typing judgments w.r.t. a finite set of names, namely those that can occur (free) in the judgment. The mathematical underpinning for names is provided by [GP99]. In particular, properties are invariant w.r.t. name permutation (equivariance), but not w.r.t. name substitution.

The syntax of MML^N_ν is abstracted over symbolic names $X \in \mathsf{Name}$, basic types b, 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 $\mathcal{X} \subseteq_{fin} \mathsf{Name}$ of names that can occur in the types and signatures.

– $\boxed{\tau \in \mathsf{T}_\mathcal{X} ::= b \mid [\Sigma|\tau] \mid M\tau}$ $\mathcal{X}$-types, where
  $\Sigma \in \Sigma_\mathcal{X} \overset{\Delta}{=} \mathcal{X} \overset{fin}{\to} \mathsf{T}_\mathcal{X}$ is a $\mathcal{X}$-signature $\{X_i : \tau_i | i \in m\}$
– $\boxed{e \in \mathsf{E} ::= x \mid \theta.X \mid e\langle\theta\rangle \mid \mathsf{b}(r)e \mid \mathsf{ret}\ e \mid \mathsf{do}\ x \leftarrow e_1; e_2 \mid \nu X.e}$ 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 (see Section 5 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 $\mathsf{b}(r)e$ *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 $\mathsf{ret}\ e$ and $\mathsf{do}\ x \leftarrow e_1; e_2$ are used to terminate and sequence computations, $\nu X.e$ generates a *fresh* name for use within the computation e.

As a simple example, let us consider the fragment `b(r)(r.X*r.X)` which can be correctly linked with 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

$$x\ \frac{}{\mathcal{X};\Pi;\Gamma\vdash x\colon\tau}\ \Gamma(x)=\tau \qquad \text{resolve}\ \frac{\mathcal{X};\Pi;\Gamma\vdash\theta\colon\Sigma}{\mathcal{X};\Pi;\Gamma\vdash\theta.X\colon\tau}\ \tau=\Sigma(X)$$

$$\text{link}\ \frac{\begin{array}{c}\mathcal{X};\Pi;\Gamma\vdash e\colon[\Sigma|\tau]\\ \mathcal{X};\Pi;\Gamma\vdash\theta\colon\Sigma'\end{array}}{\mathcal{X};\Pi;\Gamma\vdash e\langle\theta\rangle\colon\tau}\ \Sigma\subseteq\Sigma' \qquad \text{box}\ \frac{\mathcal{X};\Pi,r\colon\Sigma;\Gamma\vdash e\colon\tau}{\mathcal{X};\Pi;\Gamma\vdash \mathsf{b}(r)e\colon[\Sigma|\tau]}$$

$$r\ \frac{\Pi(r)=\Sigma}{\mathcal{X};\Pi;\Gamma\vdash r\colon\Sigma} \quad ?\ \frac{}{\mathcal{X};\Pi;\Gamma\vdash ?\colon\emptyset} \quad \text{extr}\ \frac{\begin{array}{c}\mathcal{X};\Pi;\Gamma\vdash\theta\colon\Sigma\\ \mathcal{X};\Pi;\Gamma\vdash e\colon\tau\end{array}}{\mathcal{X};\Pi;\Gamma\vdash\theta\{X\colon e\}\colon\Sigma\{X\colon\tau\}}$$

$$\text{ret}\ \frac{\mathcal{X};\Pi;\Gamma\vdash e\colon\tau}{\mathcal{X};\Pi;\Gamma\vdash \mathsf{ret}\ e\colon M\tau} \qquad \text{do}\ \frac{\begin{array}{c}\mathcal{X};\Pi;\Gamma\vdash e_1\colon M\tau_1\\ \mathcal{X};\Pi;\Gamma,x\colon\tau_1\vdash e_2\colon M\tau_2\end{array}}{\mathcal{X};\Pi;\Gamma\vdash \mathsf{do}\ x\leftarrow e_1;e_2\colon M\tau_2}$$

$$\nu\ \frac{\mathcal{X},X;\Pi;\Gamma\vdash e\colon M\tau}{\mathcal{X};\Pi;\Gamma\vdash \nu X.e\colon M\tau}\ X\notin \mathrm{FV}(\Pi,\Gamma,\tau)$$

Table 1. Type System for MML^N_ν

section is devoted to the formal definition of MML^N_ν. More interesting examples (with informal explanatory text) can be found in Section 5.

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

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

4.1 Type system

The typing judgments are $\mathcal{X};\Pi;\Gamma\vdash e\colon\tau$ (i.e. e has type τ) and $\mathcal{X};\Pi;\Gamma\vdash\theta\colon\Sigma$ (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
- $\Pi\colon\mathsf{R}\overset{fin}{\rightarrow}\Sigma_{\mathcal{X}}$ is a signature assignment $\{r_i\colon\Sigma_i|i\in m\}$ for resolver variables
- $\Gamma\colon\mathsf{X}\overset{fin}{\rightarrow}\mathsf{T}_{\mathcal{X}}$ is a type assignment $\{x_i\colon\tau_i|i\in m\}$ for term variables

The typing rules are given in Table 1. All the 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 linking of a fragment $e\colon[\Sigma|\tau]$ with a resolver θ whose signature Σ' includes Σ. All the other rules are standard.

4.2 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

(resolve) $(\theta\{X{:}\,e\}).X \longrightarrow e$
(delegate) $(\theta\{X{:}\,e\}).X' \longrightarrow \theta.X'$ if $X' \neq X$
(link) $(\mathsf{b}(r)e)\langle\theta\rangle \longrightarrow e[r{:}\,\theta]$

Simplification enjoys the following properties.

Theorem 1 (Church-Rosser). *The simplification relation* $\longrightarrow$ *is confluent.*

Theorem 2 (Subject Reduction).

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

4.3 Computation

The computation relation $Id \longmapsto Id' \mid \mathsf{done}$ is defined using evaluation contexts 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 a rule for generation of a fresh name (this is the only rule that extends the name space).

- $\boxed{E \in \mathsf{EC}{::=}\,\square \mid E[\mathsf{do}\ x \leftarrow \square; e]}$ evaluation contexts
- $(\mathcal{X}|e, E) \in \mathsf{Conf} \stackrel{\Delta}{=} \mathcal{P}_{fin}(\mathsf{Name}) \times \mathsf{E} \times \mathsf{EC}$ configurations consist of the current name space $\mathcal{X}$ (which grows as computation progresses), the program fragment e under consideration, and its evaluation context E
- $\boxed{rc \in \mathsf{RC}{::=}\,\mathsf{ret}\ e \mid \mathsf{do}\ x \leftarrow e_1; e_2 \mid \nu X.e}$ 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 (Bisimumation). *If* $Id \equiv (\mathcal{X}|e, E)$ *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* $\mathsf{Conf} \cup \{\mathsf{done}\}$*.*

Administrative steps

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

Name generation step

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

Table 2. Computation Relation

$$\square\ \frac{}{\mathcal{X}; \square\colon M\tau \vdash \square\colon M\tau} \qquad \frac{\mathcal{X}; \square\colon M\tau_2 \vdash E\colon M\tau' \quad \mathcal{X}; \emptyset; x\colon \tau_1 \vdash e\colon M\tau_2}{\mathcal{X}; \square\colon M\tau_1 \vdash E[\mathsf{do}\ x \leftarrow \square; e]\colon M\tau'}$$

Table 3. Well-formed Evaluation Contexts

4.4 Type Safety

Following Felleisen, type safety can be decomposed in two properties: subject reduction and progress. We refer to [MF03] for a formulation of these properties in the context of a monadic metalanguage.

Definition 1 (Well-formed configuration). $\vdash (\mathcal{X}|e, E)\colon \tau' \overset{\Delta}{\Longleftrightarrow} \tau' \in \mathsf{T}_\emptyset$ *and* $\exists \tau \in \mathsf{T}_\mathcal{X}$ *s.t.* $\mathcal{X}; \emptyset; \emptyset \vdash e\colon M\tau$ *and* $\mathcal{X}; \square\colon M\tau \vdash E\colon M\tau'$ *(see Table 3).*

Theorem 4 (Subject Reduction).

– *If* $\vdash Id_1\colon \tau'$ *and* $Id_1 \longrightarrow Id_2$*, then* $\vdash Id_2\colon \tau'$.
– *If* $\vdash Id_1\colon \tau'$ *and* $Id_1 \longmapsto Id_2$*, then* $\vdash Id_2\colon \tau'$.

Theorem 5 (Progress). *If* $\vdash (\mathcal{X}|e, E)\colon \tau'$*, then*

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

5 Programming examples

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

– the first exemplifies programming with *open* fragments;
– the second recasts the multi-stage programming method of [TS97] by making a simplified use of *open* fragments;
– the third uses *closed* fragments, and allows a further comparison with other calculi for run-time code generation and staging.

To improve readability we use ML-like notation for functions (β-reduction is a sound simplification in monadic metalanguages) and operations on references, 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 need for fresh name generation. In our calculus, a component is identified with a fragment of type $[\Sigma|\tau]$, where Σ specifies what information needs to be provided for deployment. Generative programming supports dynamic manufacturing of customized components from elementary (highly reusable) components. The most appropriate building block for generative programming are polymorphic functions $G{:}\,\forall p.[p, \Sigma_i|\tau_i] \to M[p, \Sigma|\tau]$ (we refer to [AM04] for a polymorphic extension of the type system). The result type of G is computational, because generation may require computational activities, while the *signature variable* p classifies the information passed to the parameters of G, but not directly used or provided in the implementation of 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}]
```

`Ac` 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. the 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 local criteria. So we need a more refined generator, with an extra parameter for computing the new balance based on the state of the account after the bank's update

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

The signature variable `p'` classifies the information needed to compute local taxes. In general `p'` and `p` are unrelated, and identifying them means that banks and local authorities rely on the same information. `TaxedAc` is defined 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'>}>)};
```

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 $G\colon [\Sigma_i|\tau_i] \to M[\Sigma|\tau]$ 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 recast the *multi-stage programming method* of [TS97] (see also [CMS03]) using the power function, which is a classical example for staged programming.

1. The method starts from a "conventional" program exp with two parameters. In the specific example, exp takes an exponent n, a base x, and then computes x^n by making recursive calls

   ```
   fun exp n x = if n=0 then ret(1.0)
           else do {y <- (exp (n-1) x); ret(x*y)};
   >   exp = ... : int -> real -> M real
   ```

   The result type of exp is computational, because we consider recursion a computational effect.
2. Then one obtains a "staged" version exp_a, which replaces the second parameter (the base x) with an open fragment u, and builds an open fragment representing the desired result (i.e. x^n)

   ```
   fun exp_a n u = if n=0 then ret(b(r) 1.0)
             else do {v <- exp_a (n-1) u; ret(b(r) u<r>*v<r>)};
   >   exp_a = ... : int -> [p|real] -> M[p|real]
   ```

   The staged version is polymorphic in the signature variable p.
3. By exploiting the polymorphism of exp_a, one defines a *code generator* exp_cg. Given the base n, the generator calls exp_a with a "dummy" parameter b(r) r.X, then builds an open fragment representing a function

   ```
   fun exp_cg n = nu X. do {v <- exp_a n (b(r) r.X);
                            ret (b(r) fn x => v<r{X:x}>)};
   >   exp_cg = ... : int -> M[p|real -> real]
   ```

   The type of exp_cg says that recursion is unfolded at "specialization" time, when the exponent n is known.
4. By instantiating p with the empty signature, one gets an *optimized* program

   ```
   fun exp_o n = do {v <- exp_cg n; ret(v<?>)};
   >   exp_o = ... : int -> M(real -> real)
   ```

   The type of exp_o differs from the type of the conventional program exp to reflect the different timing in unfolding recursion. Namely, exp_o unfolds the recursion when the parameter n is known. For instance, when $n = 2$

   ```
   do  sq_o <- exp_o 2;
   >   sq_o = (fn x => x*(x*1.0)) : real -> real
   ```

The multi-stage programming method makes use of open fragments of type $[p|\tau]$ (these types are similar to the code types annotated with environment classifiers $\langle\tau\rangle^\alpha$ used by [TN03,CMT04]). One can easily recast the multi-stage programming method also in the presence of more complex computational effects (while in MetaML there are typing problems). For instance, when the conventional program is an imperative variant `p:int->real->(real ref)->M unit` of `exp`

```
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` takes an exponent n, a base x and a reference `y`, then it initializes y with 1.0 and repeatedly multiplies the content of y with x until it becomes x^n. The "staged" version, `p_a`, is defined in the obvious way, and its type says that some computations are postponed to the second stage

```
fun p_a n u v= if n=0 then ret(b(r) v<r>:=1.0)
        else do {
             w <- p_a (n-1) u v;
             ret(b(r) do {w<r>; y' <-!v<r>; v<r>:=u<r>*y'})};
>   p_a = ... : int ->  [p|real] -> [p|Ref real] -> M[p|M unit]
```

In comparison to MetaML, we don't face the problems due to execution of *potentially open* code or *scope extrusion*, which motivated the introduction of closed types in [CMS03]. The reason is that in MML_ν^N one has a better control of the name space and name resolution.

Example 3. We reconsider the power function `exp:int->real->M real`, and give an alternative way to define `exp_o:int-> M(real-> real)`, which does not involve fresh name generation.

```
(* conventional program *)
fun exp n x = if n=0 then ret(1.0)
        else do {y <- (exp (n-1) x); ret(x*y)};
>   exp = ... : int -> real -> M real
(* staged program *)
fun exp_a n u = if n=0 then ret(b(r) 1.0)
          else do {v <- exp_a (n-1) u; ret(b(r) u<r>*v<r>)};
>   exp_a = ... : int -> [p|real] -> M[p|real]
(* exp_c generates a fragment with hook X for base *)
fun exp_c n = do {v <- exp_a n (b(r) r.X);
                  ret (b(r) fn x => v<r>)};
>   exp_c = ... : int -> M[X:real|real]
(* optimized program *)
fun exp_o n = do {v <- exp_c n; ret(fn x => u<?{X:x}>)};
>   exp_o = ... : int -> M(real -> real)
```

The definition of `exp_c` relies on a pre-existing name `X`, while `exp_cg` uses a freshly generated name. MetaML does not allow to mention names explicitly, thus it has no analogue of `exp_c` nor of the type `[X:real|real]`. On the other hand, $\nu^{\Box}$ has an analogue of `exp_c` (see `exp'` in [NPar, Example 2]), but the name `X` has to be declared of type `real` globally.

6 Relating MML_{ν}^{N} to MetaML

In this section we define a monadic CBV translation of a 2-level version of MetaML into MML_{ν}^{N} (extended with functional types), and show that the translation preserves the operational semantics. We make no formal claim about preservation of typing, since we have not been able to extend the translation to types. We have not defined a monadic CBV translation of the whole MetaML, since key ideas would get confused with orthogonal issues involved in the translation of a multi-level language. Restricting to a 2-level language allows to bring these ideas in the foreground.

6.1 MetaML_2

We give the formal definition (syntax, a simplified type system and big-step CBV operational semantics) of MetaML_2, a 2-level version of MetaML. As customary for 2-level languages, the syntax (type system and operational semantics) is stratified in two levels: the meta-level 0, and the object-level 1.

- $\boxed{\begin{array}{l}\tau^0 \in \mathsf{T}^0 ::= b \mid \tau_1^0 \to \tau_2^0 \mid \langle \tau^1 \rangle \\ \tau^1 \in \mathsf{T}^1 ::= b \mid \tau_1^1 \to \tau_2^1\end{array}}$ types at level 0 and 1, note that $\mathsf{T}^1 \subset \mathsf{T}^0$
- $\boxed{\begin{array}{l}e^0 \in \mathsf{E}^0 ::= \underline{v^0} \mid e_1^0 e_2^0 \mid \langle e^1 \rangle \mid \mathsf{run}\ e^0 \\ v^0 \in \mathsf{V}^0 ::= x^0 \mid \lambda x^0.e^0 \mid \langle v^1 \rangle \\ e^1 \in \mathsf{E}^1 ::= \underline{v^1} \mid \lambda x^1.e^1 \mid e_1^1 e_2^1 \mid \tilde{\ } e^0 \\ v^1 \in \mathsf{V}^1 ::= x^1 \mid \lambda x^1.v^1 \mid v_1^1 v_2^1\end{array}}$ terms and values at level 0 and 1

We give an informal semantics of the language.

- A value v^1 corresponds to an object-level program, while a term e^1 may require some meta-level computation to get an object-level program.
- The type $\langle \tau^1 \rangle$ classifies code, i.e. meta-level values of the form $\langle v^1 \rangle$ representing an object-level program v^1. Brackets $\langle e^1 \rangle$ and escape $\tilde{\ } e^0$ allow to move between meta-level code and the corresponding object-level program, in particular $\tilde{\ }\langle e^1 \rangle$ and e^1 evaluate to the same object-level program.
- The construct $\mathsf{run}\ e^0$ first evaluates e^0 to code $\langle v^1 \rangle$, then evaluates the object-level program v^1 (provided it is a *complete* program, i.e. $\mathrm{FV}(v^1) = \emptyset$).

Besides the stratification in two levels, there are the following syntactic differences between MetaML_2 and MetaML of [CMT04]:

- MetaML_2 values are explicitly marked in terms. This avoids re-evaluation and a general pitfall of monadic CBV translations, namely preservation of the operational semantics (as stated in Theorem 6) would fail.

$$\frac{}{\underline{v^n} \xhookrightarrow{n} v^n} \qquad \frac{e_1^0 \xhookrightarrow{0} \lambda x^0.e^0 \quad e_2^0 \xhookrightarrow{0} v^0 \quad e^0[x^0\colon v^0] \xhookrightarrow{0} v_1^0}{e_1^0 e_2^0 \xhookrightarrow{0} v_1^0}$$

$$\frac{e^1 \xhookrightarrow{1} v^1}{\langle e^1\rangle \xhookrightarrow{0} \langle v^1\rangle} \qquad \frac{e^0 \xhookrightarrow{0} \langle v^1\rangle \quad v^1 \downarrow \xhookrightarrow{0} v^0}{\mathsf{run}\ e^0 \xhookrightarrow{0} v^0}\ \mathrm{FV}(v^1) = \emptyset$$

$$\frac{e^1 \xhookrightarrow{1} v^1}{\lambda x^1.e^1 \xhookrightarrow{1} \lambda x^1.v^1} \qquad \frac{e_1^1 \xhookrightarrow{1} v_1^1 \quad e_2^1 \xhookrightarrow{1} v_2^1}{e_1^1 e_2^1 \xhookrightarrow{1} v_1^1 v_2^1} \qquad \frac{e^0 \xhookrightarrow{0} \langle v^1\rangle}{\tilde{}e^0 \xhookrightarrow{1} v^1}$$

where demotion $v^1 \downarrow$ is defined by induction on v^1

$$x^1 \downarrow = \underline{x^0} \qquad (\lambda x^1.v^1) \downarrow = \underline{\lambda x^0.v^1 \downarrow} \qquad (v_1^1 v_2^1) \downarrow = v_1^1 \downarrow v_2^1 \downarrow$$

Table 4. Big-Step Operational Semantics for MetaML_2

– Cross-stage persistence, i.e. the ability to include meta-level values ($v^0\colon \tau^1$) into object-level programs, is excluded from MetaML_2. This simplifies some definitions and technical lemmas.
– In MetaML_2 code types are not annotated with environment classifiers.

A type system (without the environment classifiers of [TN03,CMT04]) is given by the following rules, where n ranges over levels (i.e. is either 0 or 1):

$$x^n\ \frac{\Gamma(x^n) = \tau^n}{\Gamma \vdash_n x^n\colon \tau^n} \qquad \mathrm{val}\ \frac{\Gamma \vdash_n v^n\colon \tau^n}{\Gamma \vdash_n \underline{v^n}\colon \tau^n} \qquad \mathrm{brk}_v\ \frac{\Gamma \vdash_1 v^1\colon \tau^1}{\Gamma \vdash_0 \langle v^1\rangle\colon \langle \tau^1\rangle}$$

$$\lambda^n\ \frac{\Gamma, x^n\colon \tau_1^n \vdash_n e^n\colon \tau_2^n}{\Gamma \vdash_n \lambda x^n.e^n\colon \tau_1^n \to \tau_2^n} \qquad @^n\ \frac{\Gamma \vdash_n e_1^n\colon \tau_1^n \to \tau_2^n \quad \Gamma \vdash_n e_2^n\colon \tau_1^n}{\Gamma \vdash_n e_1^n e_2^n\colon \tau_2^n}$$

$$\lambda_v\ \frac{\Gamma, x^1\colon \tau_1^1 \vdash_1 v^1\colon \tau_2^1}{\Gamma \vdash_1 \lambda x^1.v^1\colon \tau_1^1 \to \tau_2^1} \qquad @_v\ \frac{\Gamma \vdash_1 v_1^1\colon \tau_1^1 \to \tau_2^1 \quad \Gamma \vdash_1 v_2^1\colon \tau_1^1}{\Gamma \vdash_1 v_1^1 v_2^1\colon \tau_2^1}$$

$$\mathrm{run}\ \frac{\Gamma \vdash_0 e^0\colon \langle \tau^1\rangle}{\Gamma \vdash_0 \mathsf{run}\ e^0\colon \tau^1} \qquad \mathrm{brk}\ \frac{\Gamma \vdash_1 e^1\colon \tau^1}{\Gamma \vdash_0 \langle e^1\rangle\colon \langle \tau^1\rangle} \qquad \mathrm{esc}\ \frac{\Gamma \vdash_0 e^0\colon \langle \tau^1\rangle}{\Gamma \vdash_1 \tilde{}e^0\colon \tau^1}$$

The operational semantics of Table 4 consists of two relations $e^n \xhookrightarrow{n} v^n$, that evaluate terms to values. The operational semantics uses only the substitution $\_[x^0\colon v^0]$ and enjoys the following properties.

Proposition 1 (Operational properties).

– demote $\dfrac{\{x_i^1\colon \tau_i^1 | i \in m\} \vdash_1 v^1\colon \tau^1}{\{x_i^0\colon \tau_i^1 | i \in m\} \vdash_0 v^1 \downarrow\colon \tau^1}$

– SR $\dfrac{\Gamma \vdash_n e^n\colon \tau^n \quad e^n \xhookrightarrow{n} v^n}{\Gamma \vdash_n v^n\colon \tau^n}$

e^0	$\llbracket e^0 \rrbracket^\rho$
$\underline{v^0}$	$\mathsf{ret}\ \llbracket v^0 \rrbracket^\rho$
$e^0_1 e^0_2$	$\mathsf{do}\ x_1 \leftarrow \llbracket e^0_1 \rrbracket^\rho; x_2 \leftarrow \llbracket e^0_2 \rrbracket^\rho; x_1 x_2$
$\langle e^1 \rangle$	$\llbracket e^1 \rrbracket^\rho$
$\mathsf{run}\ e^0$	$\mathsf{do}\ x \leftarrow \llbracket e^0 \rrbracket^\rho; x\langle ? \rangle$
v^0	$\llbracket v^0 \rrbracket^\rho$
x^0	e where $e = \rho(x^0)$
$\lambda x^0.e^0$	$\lambda x.\llbracket e^0 \rrbracket^{\rho, x^0 : x}$
$\langle v^1 \rangle$	$\mathsf{b}(r)\llbracket v^1 \rrbracket^\rho_r$
e^1	$\llbracket e^1 \rrbracket^\rho$
$\underline{v^1}$	$\mathsf{ret}\ (\mathsf{b}(r)\llbracket v^1 \rrbracket^\rho_r)$
$e^1_1 e^1_2$	$\mathsf{do}\ x'_1 \leftarrow \llbracket e^1_1 \rrbracket^\rho; x'_2 \leftarrow \llbracket e^1_2 \rrbracket^\rho; \mathsf{ret}\ (\mathsf{b}(r)\mathsf{do}\ x_1 \leftarrow x'_1\langle r \rangle; x_2 \leftarrow x'_2\langle r \rangle; x_1 x_2)$
$\lambda x^1.e^1$	$\nu X.\mathsf{do}\ x' \leftarrow \llbracket e^1 \rrbracket^{\rho, x^1 : \mathsf{b}(r) r.X}; \mathsf{ret}\ (\mathsf{b}(r)\mathsf{ret}\ \lambda x.x'\langle r\{X : x\}\rangle)$
$\tilde{}e^0$	$\llbracket e^0 \rrbracket^\rho$
v^1	$\llbracket v^1 \rrbracket^\rho_\theta$
x^1	$\mathsf{ret}\ e[r : \theta]$ where $\mathsf{b}(r)e = \rho(x^1)$
$\lambda x^1.v^1$	$\mathsf{ret}\ \lambda x.\llbracket v^1 \rrbracket^{\rho, x^1 : \mathsf{b}(r) x}_\theta$
$v^1_1 v^1_2$	$\mathsf{do}\ x_1 \leftarrow \llbracket v^1_1 \rrbracket^\rho_\theta; x_2 \leftarrow \llbracket v^1_2 \rrbracket^\rho_\theta; x_1 x_2$

Table 5. Translation of MetaML$_2$ terms and values

6.2 Translation of MetaML$_2$ into MML$^N_\nu$

Table 5 defines (by induction on the syntax of MetaML$_2$) a translation $\llbracket e^n \rrbracket^\rho$, $\llbracket v^0 \rrbracket^\rho$ and $\llbracket v^1 \rrbracket^\rho_\theta$, where θ is a resolver and ρ is a (partial) mapping from variables of MetaML$_2$ to terms of MML$^N_\nu$ such that

- a meta-level variable x^0 is mapped to a term e
- an object-level variable x^1 is mapped to a fragment $\mathsf{b}(r)e$

The parameters ρ and θ are convenient to state some properties (see Lemma 1), but for the definition of the translation it suffices to take $\theta = r$ and $\rho(x^0) = x$.

Some clauses in the definition of the translation deserve to be commented:

- terms of the form $\underline{v^n}$ are always translated into terms of the form $\mathsf{ret}\ e$, since values v^n do not require meta-level computation
- the translations of $\langle e^1 \rangle$ and e^1 are the same (and similarly for $\tilde{}e^0$ and e^0), because the bijection between meta-level code and object-level programs is collapsed to an equality
- the translation of $\langle v^1 \rangle$ is a fragment, which results into an object-level program after linking, thus the translation of v^1 depends on a resolver θ
- the translations of $e^1_1 e^1_2$ and $\lambda x^1.e^1$ are meta-level computations to generate code representing an application and abstraction in the object language

– the translation of values at level 1 (i.e. object-level programs) is like the monadic CBV translation of the λ-calculus, as the object language is CBV.

There are problems in extending the translation to types (thus we make no formal claim about preservation of typing). More precisely, the problem is to identify a signature to replace of ... in the following inductive definition

$$[\![b]\!] = b \qquad [\![\tau_1^n \to \tau_2^n]\!] = [\![\tau_1^n]\!] \to M[\![\tau_2^n]\!] \qquad [\![\langle\tau^1\rangle]\!] = [\ldots|M[\![\tau^1]\!]]$$

The translation preserves the operational semantics in the following sense:

Theorem 6. *$e^0 \overset{0}{\hookrightarrow} v^0$ and $\mathrm{FV}(e^0) = \emptyset$ imply $(\emptyset|[\![e^0]\!], \Box) \overset{*}{\Longrightarrow} (\mathcal{X}|\mathsf{ret}\ [\![v^0]\!], \Box)$ for some $\mathcal{X}$, where $\Longrightarrow\ \overset{\Delta}{\equiv}\ \longrightarrow \cup \longmapsto$.*

The result is a consequence of the following lemmas (stated without proof).

Lemma 1 (Properties of Translation).

1. *If $e = [\![v^0]\!]^\rho$, then $[\![\_[x^0\colon v^0]]\!]^\rho = [\![\_]\!]^{\rho,x^0:e}$ and $[\![\_[x^0\colon v^0]]\!]^\rho_\theta = [\![\_]\!]^{\rho,x^0:e}_\theta$*
2. *$[\![v^1]\!]^\rho_\theta = [\![v^1\downarrow]\!]^{\rho'}$, if $\rho'(x^0) = e[r\colon\theta]$ when $\rho(x^1) = \mathsf{b}(r)e$ and $x^1 \in \mathrm{FV}(v^1)$*
3. *$[\![v^1]\!]^{\rho_1}_{\theta_1} \overset{*}{\longrightarrow} [\![v^1]\!]^{\rho_2}_{\theta_2}$, if $e_1[r\colon\theta_1] \overset{*}{\longrightarrow} e_2[r\colon\theta_2]$ when $\rho_i(x^1) = \mathsf{b}(r)e_i$ and $x^1 \in \mathrm{FV}(v^1)$.*

Lemma 2 (Preservation of Evaluation). *For any $\mathcal{X}$ and E*

– *$e^0 \overset{0}{\hookrightarrow} v^0$ implies $(\mathcal{X}|[\![e^0]\!]^\rho, E) \overset{*}{\Longrightarrow} (\mathcal{X}'|\mathsf{ret}\ [\![v^0]\!]^\rho, E)$ for some $\mathcal{X}'$*
– *$e^1 \overset{1}{\hookrightarrow} v^1$ implies $(\mathcal{X}|[\![e^1]\!]^\rho, E) \overset{*}{\Longrightarrow} (\mathcal{X}'|\mathsf{ret}\ \mathsf{b}(r)[\![v^1]\!]^\rho_r, E)$ for some $\mathcal{X}'$*

provided $x^1 \in \mathrm{FV}(e^n)$ implies $\rho(x^1) = \mathsf{b}(r)r.X$for some $X \in \mathcal{X}$.

We conclude with three examples of MetaML_2- terms. Each term evaluates to the same MetaML_2-value $v^0 \overset{\Delta}{\equiv} \langle\lambda x^1.x^1\rangle$, i.e. the code representing the object-level identity function. However, the MML^N_ν-translation of these terms reflect the different complexity of the evalution to v^0.

– The term $e^0_0 \overset{\Delta}{\equiv} \underline{v^0}$ evaluates immediately to v^0.
  The translation $[\![v^0]\!]^\emptyset$ is given by $e \overset{\Delta}{\equiv} \mathsf{b}(r)\mathsf{ret}\ \lambda x.\mathsf{ret}\ x$, where $\mathsf{ret}\ \lambda x.\mathsf{ret}\ x$ is the CBV monadic translation of $\lambda x.x$. The translation $[\![e^0_0]\!]^\emptyset$ is simply $\mathsf{ret}\ e$.
  Therefore, Theorem 6 holds trivially for $e^0_0 \overset{0}{\hookrightarrow} v^0_0$.
– The term $e^0_1 \overset{\Delta}{\equiv} \langle\lambda x^1.\underline{x^1}\rangle$ evaluates to v^0, but the evaluation steps are more complex. This complexity is mirrored in the translation $[\![e^0_1]\!]^\emptyset$ given by
  $$\begin{aligned} e_1 \overset{\Delta}{\equiv} \nu X.\mathsf{do}\ & x' \leftarrow \mathsf{ret}\ (\mathsf{b}(r)\mathsf{ret}\ r.X);\\ & \mathsf{ret}\ (\mathsf{b}(r)\mathsf{ret}\ \lambda x.x'\langle r\{X\colon x\}\rangle)\end{aligned}$$
  Theorem 6 for $e^0_1 \overset{0}{\hookrightarrow} v^0$ yields $(\emptyset|e_1, \Box) \overset{+}{\Longrightarrow} (X|\mathsf{ret}\ e, \Box)$.

– The term $e_2^0 \overset{\Delta}{\equiv} \langle \lambda x^1. \tilde{~}((\lambda x^0.x^0)\underline{\langle x^1 \rangle}) \rangle$ evaluates to v^0, and requires evaluation within the body of the λx^1-binder. The translation $[\![e_2^0]\!]^{\emptyset}$ is given by

$$\begin{aligned} e_2 \overset{\Delta}{\equiv} \nu X.\mathsf{do}\ x' \leftarrow \mathsf{do}\ & x_1 \leftarrow \mathsf{ret}\ (\lambda x.\mathsf{ret}\ x); \\ & x_2 \leftarrow \mathsf{ret}\ (\mathsf{b}(r)\mathsf{ret}\ r.X); \\ & x_1 x_2; \\ \mathsf{ret}\ (\mathsf{b}(r)\mathsf{ret}\ & \lambda x.x'\langle r\{X\colon x\}\rangle) \end{aligned}$$

Theorem 6 yields $(\emptyset | e_2, \square) \overset{+}{\Longrightarrow} (X | \mathsf{ret}\ e, \square)$, but the steps needed to reach the final configuration are strictly more than in the previous case.

7 Relating MML_{ν}^{N} to CMS

In this section we recall CMS [AZ02], a purely functional calculus of mixin modules, and introduce ML_{Σ}^{N}, a variant of MML_{ν}^{N}. Then we define a translation of CMS in ML_{Σ}^{N} preserving CMS typing and simplification up to Ariola's equational axioms [AB02] for recursion. We summarize the main differences between MML_{ν}^{N} and CMS (for those already familiar with CMS).

– CMS has a fixed infinite set of names (but a program uses only finitely many of them) and no fresh name generation facility.
– CMS is a pure calculus, thus we can restrict to the fragment of MML_{ν}^{N} without computational types, 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 (comparable to the λ-calculus) even without recursion, and one can add recursion following standard approaches.

7.1 CMS

We recall the calculus of mixin modules CMS, and refer to [AZ99,AZ02] for further details. The syntax of CMS is abstracted over symbolic names $X \in \mathsf{Name}$, and term variables $x \in \mathsf{X}$. For simplicity, we avoid to introduce core terms and types (in [AM04] the calculus is parametrized w.r.t. a core calculus).

– $\boxed{\tau \in \mathsf{CMST} ::= [\Sigma_1; \Sigma_2]}$ types, where $\Sigma\colon \mathsf{Name} \overset{fin}{\to} \mathsf{CMST}$
– $\boxed{E \in \mathsf{CMSE} ::= x \mid [\iota; o; \rho] \mid E_1 + E_2 \mid E \setminus X \mid E!X \mid E.X}$ terms, where $\iota\colon \mathsf{X} \overset{fin}{\to} \mathsf{Name}$, $o\colon \mathsf{Name} \overset{fin}{\to} \mathsf{CMSE}$, $\rho\colon \mathsf{X} \overset{fin}{\to} \mathsf{CMSE}$ and we implicitly require that $\mathsf{dom}(\iota) \# \mathsf{dom}(\rho)$ for well-formed of $[\iota; o; \rho]$.

Free variables are defined as follows (omitting trivial cases): $\mathrm{FV}([\iota; o; \rho]) = (\mathrm{FV}(o) \cup \mathrm{FV}(\rho)) \setminus (\mathsf{dom}(\iota) \cup \mathsf{dom}(\rho))$. Thus one can freely rename the bound variables in $dom(\iota) \cup \mathsf{dom}(\rho)$, as done implicitly in the reduction rule (sum) below. We first give an informal overview of the calculus:

- The type $[\Sigma_1; \Sigma_2]$ specifies the names and types of the *deferred* (Σ_1) and *defined* (Σ_2) components of a mixin. The deferred components can be referred in the mixin, but are not defined, therefore they need to be resolved (see the freeze operation described below). The defined components corresponds to the exported definitions of the mixin.
- Term variables are used for local referencing of components, whereas names are needed for dealing with global access and linking of components. As in MML^N_ν, names are not terms.
- In a basic mixin $[\iota; o; \rho]$, ι specifies the deferred components. The mapping to names is needed for component resolution (see the freeze operation described below). The defined components (o) are associated with names, whereas local components (ρ) are introduced by variables and can be mutually recursive.
- The sum operation ($E_1 + E_2$) performs the union of the deferred components (in the sense that components with the same name are shared), and the disjoint union of the defined and local components of the two mixins.
- The *freeze* operation ($E!X$) binds the deferred component X to the expression of the defined component X in the same mixin; in this way a name can be resolved, and a deferred component becomes local. Cross-module recursion is obtained as a combination of the sum and the freeze operations.
- The *delete* operation ($E \setminus X$) is used for hiding defined components.
- Selection of a defined component ($E.X$) is only allowed for mixin with no deferred components.

Typing rules The typing judgment has form $\Gamma \vdash_{\mathsf{CMS}} E\colon \tau$, where $\Gamma\colon \mathsf{X} \overset{fin}{\to} \mathsf{CMST}$. The typing rules are given in Table 6, where two signatures Σ_1 and Σ_2 are *compatible* iff $\Sigma_1(X) = \Sigma_2(X)$ for all $X \in \mathsf{dom}(\Sigma_1) \cap \mathsf{dom}(\Sigma_2)$.

Simplification rules We define the relation $\xrightarrow[\mathsf{CMS}]{}$ as the compatible closure of the simplification rules defined in Table 7.

7.2 ML^N_Σ

The syntax of ML^N_Σ is defined in two steps. First, we remove from MML^N_ν computational types (and consequently monadic operations, like $\nu X.e$). In the resulting calculus, called ML^N, the computation relation disappears (CMS is a pure calculus), and $\mathcal{X}$ could be left implicit in the typing judgments $\mathcal{X}; \Pi; \Gamma \vdash e\colon \tau$, since the typing judgments 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 $\Sigma \in \mathsf{\Sigma}_{\mathcal{X}} \overset{\Delta}{=} \mathcal{X} \overset{fin}{\to} \mathsf{T}_{\mathcal{X}}$ is a $\mathcal{X}$-signature
- $\boxed{e \in \mathsf{E} += o \mid e.X \mid e_1 + e_2 \mid e \setminus X \mid \mathsf{let}\ \rho\ \mathsf{in}\ e}$ terms, where
  $o\colon \mathsf{Name} \overset{fin}{\to} \mathsf{E}$ is a record $\{X_i\colon e_i | i \in m\}$ and
  $\rho\colon \mathsf{X} \overset{fin}{\to} \mathsf{E}$ is a (recursive) binding $\{x_i\colon e_i | i \in m\}$.

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

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

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

$$\text{freeze}\ \frac{\Gamma \vdash_{\mathsf{CMS}} e{:}\,[\Sigma_1;\Sigma_2]}{\Gamma \vdash_{\mathsf{CMS}} e!X{:}\,[\Sigma_1\setminus X;\Sigma_2]}\ \tau=\Sigma_1(X)=\Sigma_2(X)$$

$$\text{select}\ \frac{\Gamma \vdash_{\mathsf{CMS}} e{:}\,[\emptyset;\Sigma]}{\Gamma \vdash_{\mathsf{CMS}} e.X{:}\,\tau}\ \tau=\Sigma(X)$$

Table 6. Type System for CMS

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

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

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

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

Table 7. Simplification rules for CMS

Free variables are defined as follows (omitting trivial cases): FV(let ρ in e) $=$ FV$(e)\setminus\mathsf{dom}(\rho)$.

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 with 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 used to encode 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 but functions (with finite domain).

Table 8 gives the typing rules for the new constructs. The properties of the type system in Section 4 extend in the obvious way to ML_Σ^N. We define simplification $\longrightarrow$ for ML_Σ^N as the compatible closure of the simplification

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

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

$$\text{delete}\ \frac{\mathcal{X};\Pi;\Gamma\vdash e{:}\,\Sigma}{\mathcal{X};\Pi;\Gamma\vdash e\setminus X{:}\,\Sigma\setminus X}$$

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

Table 8. Additional Typing Rules for ML^N_Σ

rules for MML^N_ν (see Section 4.2) 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) $\mathsf{let}\ \rho\ \mathsf{in}\ e \Longrightarrow e[x{:}\,\mathsf{let}\ \rho\ \mathsf{in}\ \rho(x)\mid x\in\mathsf{dom}(\rho)]$

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

7.3 Translation of CMS into ML^N_Σ

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 9 gives the translation of CMS in ML^N_Σ. The translation can be easily extended to core terms (see [AM04]).

In the translation of a basic mixin $[\iota;o;\rho]$ the deferred variables x ($x\in\mathsf{dom}(\iota)$) are replaced with the resolution $r.X$ of the corresponding name $X=\iota(x)$, whereas the local variables x ($x\in\mathsf{dom}(\rho)$) 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 8).

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.

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

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

Table 9. Translation of CMS in ML^N_Σ

The typing preservation property of the translation can be proved

Theorem 7 (Typing preservation). *If $\Gamma \vdash_{\mathsf{CMS}} E\colon\tau$, then $\mathcal{X};\emptyset;\Gamma' \vdash E'\colon\tau'$, where $\mathcal{X}$ includes all names occurring in the derivation of $\Gamma \vdash_{\mathsf{CMS}} E\colon\tau$.*

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

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

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 8 it suffices to consider the following contexts: $\boxed{\mathsf{C}[\ ]{::=}\ \square + e \mid e + \square \mid \square \setminus X \mid \square.X}$.

The (ext-merge) and (int-merge) axioms are Ariola's merge axioms, 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 simplification up to $=_{S\cup R}$ (i.e. the congruence induced by the axioms in $S \cup R$).

Theorem 8 (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 simplifications are mapped to plain ML^N_Σ simplifications.

We conclude this section with some examples of CMS reductions and their translations in ML^N_Σ.

Example 4. Consider the **plus** reduction $E_1 \xrightarrow[\mathsf{CMS}]{} E_2$, where
$E_1 \equiv [\{x\colon A\}; \{R\colon x\}; \emptyset] + [\emptyset; \{A\colon y\}; \emptyset]$,
$E_2 \equiv [\{x\colon A\}; \{A\colon y, R\colon x\}; \emptyset]$.

The translations of E_1 and E_2 are given by
$E'_1 \equiv \mathsf{b}(r)(\mathsf{b}(r_1)(\mathsf{let}\ \emptyset\ \mathsf{in}\ \{R\colon r_1.A\}))\langle r\rangle + (\mathsf{b}(r_2)\mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y\})\langle r\rangle$
$E'_2 \equiv \mathsf{b}(r)\mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon r.A\}$.
By repeatedly applying simplifications and equational axioms in R we get
$E'_1 \xrightarrow{*}$ (by **link**)
$\mathsf{b}(r)(\mathsf{let}\ \emptyset\ \mathsf{in}\ \{R\colon r.A\}) + \mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y\} =_R$ (by **lift**)
$\mathsf{b}(r)\mathsf{let}\ \emptyset\ \mathsf{in}\ \mathsf{let}\ \emptyset\ \mathsf{in}\ \{R\colon r.A\} + \{A\colon y\} =_R$ (by **ext-merge**)
$\mathsf{b}(r)\mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon r.A\} \equiv E'_2$.

Example 5. Consider the **freeze** reduction $E_3 \xrightarrow[\mathsf{CMS}]{} E_4$, where
$E_3 \equiv E_2!A$ with E_2 defined as in the previous example,
$E_4 \equiv [\emptyset; \{A\colon y, R\colon x\}; \{x\colon y\}]$.

The translations of E_3 and E_4 are given by
$E'_3 \equiv \mathsf{b}(r)\mathsf{let}\ \{x_1\colon x_2.A, x_2\colon E'_2\langle r\{A\colon x_1\}\rangle\}\ \mathsf{in}\ x_2$
$E'_4 \equiv \mathsf{b}(r)\mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ \{A\colon y, R\colon x\}$.
By repeatedly applying simplifications and equational axioms in R we get
$E'_3 \longrightarrow$ (by **link**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon x_2.A, x_2\colon \mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon (r\{A\colon x_1\}).A\}\}\ \mathsf{in}\ x_2 \longrightarrow$ (by **resolve**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon x_2.A, x_2\colon \mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon x_1\}\}\ \mathsf{in}\ x_2 =_R$ (by **sub**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon (\mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon x_1\}).A\}\ \mathsf{in}\ \mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon x_1\} =_R$ (by **ext-merge**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon (\mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon x_1\}).A\}\ \mathsf{in}\ \{A\colon y, R\colon x_1\} =_R$ (by **lift**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon \mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon x_1\}.A\}\ \mathsf{in}\ \{A\colon y, R\colon x_1\} =_R$ (by **int-merge**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon \{A\colon y, R\colon x_1\}.A\}\ \mathsf{in}\ \{A\colon y, R\colon x_1\} \longrightarrow$ (by **resolve**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon y\}\ \mathsf{in}\ \{A\colon y, R\colon x_1\}$ which is α-equivalent to E'_4.

Example 6. Consider the **select** reduction $E_5 \xrightarrow[\mathsf{CMS}]{} E_6$, where
$E_5 \equiv E_4.R$ with E_4 defined as in the previous example,
$E_6 \equiv [\emptyset; \{X\colon y\}; \{x\colon y\}].X$.

The translations of E_5 and E_6 are given by
$E'_5 \equiv (\mathsf{b}(r)\mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ \{A\colon y, R\colon x\})\langle ?\rangle.R$,
$E'_6 \equiv (\mathsf{b}(r)\mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ \{X\colon y\})\langle ?\rangle.X$.
By repeatedly applying simplifications we get
$E'_5 \longrightarrow$ (by **link**)
$(\mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ \{A\colon y, R\colon x\}).R \longrightarrow$ (by **unfolding**)
$\{A\colon y, R\colon \mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ y\}.R \longrightarrow$ (by **resolve**)
$\mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ y \longrightarrow$ (by **unfolding**)
y.

Analogously, $E'_6 \xrightarrow{*} y$, therefore $E'_5 =_S E'_6$. Unlike the other examples, there is no way to get from E'_5 to E'_6 by applying simplifications and equations axioms in R. This explains the use of $=_{S\cup R}$ in stating Theorem 8.

8 Conclusions and related work

This section compares MML^N_ν with the CBV calculi FreshML and $\nu^\Box$. First, we recall briefly the main features of these two calculi, then we make a critical assessment based on a comparison with MML^N_ν.

– FreshML of [SPG03][2] is an extension of ML, based on a solid mathematical theory [GP99], that provides a convenient support for meta-programming. Namely, in FreshML abstract representations (i.e. modulo α-conversion) of object-level syntax co-exist with pattern matching facilities (similar to those usable on concrete parse trees) to analyse these representations.
– $\nu^\Box$ of [Nan02,NPar] is a refinement of $\lambda^\Box$ [DP01], which provides better support for symbolic manipulation by exploiting some features of FreshML (the calculus presented in [NPar] does not support program analysis). 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). MetaML has similar aims, but adopts the opposite strategy, i.e. it starts from $\lambda^\bigcirc$, nor does it build on top of FreshML (names are not part of the MetaML syntax).

If we ignore the different styles for describing the operational semantics of FreshML (a CBV evaluation relation), $\nu^\Box$ (a CBV small-step reduction relation) and MML^N_ν (a simplification and a computation relation), the key differences are:

– FreshML supports program transformation, in particular the analysis of object-level programs represented as values of an inductive datatype involving *abstraction* types $\langle\mathsf{name}\rangle\tau$.
– $\nu^\Box$ of [NPar] supports only program generation, (object-level) programs e of type τ with unresolved names in $\mathcal{X}$ are represented by values of type $\Box_\mathcal{X}\tau$. The typing rules for $\Box_\mathcal{X}\tau$ are fairly restrictive, because $\mathcal{X}$ has to include **all** unresolved names in e (and e should not contain free variables x).
– MML^N_ν supports only program generation, (object-level) programs e of type τ abstracted w.r.t. a name resolver r of signature Σ are represented by terms of type $[\Sigma|\tau]$. The typing rules for $[\Sigma|\tau]$ are similar to those for a functional type $\Sigma \to \tau$, in particular e may use other name resolvers besides r.

MML^N_ν *versus FreshML.* Typing judgments of FreshML have the standard format $\Gamma \vdash e\colon \tau$, because names do not occur in types (and in particular there are no resolvers and no signatures for typing resolvers).

In FreshML names are terms (and there is a type name 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^N_ν 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. FreshML, unlike MML^N_ν, supports the manipulation of object-level syntax modulo α-conversion. This is possible because FreshML has:

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

- an equality type name of names
- abstraction types $\langle\mathsf{name}\rangle\tau$ classifying equivalence classes of pairs (X, e) modulo renaming of X with names fresh for e (of type τ), terms $\langle e_1\rangle e_2$ to form name abstractions, and patterns $\langle x\rangle p$ to deconstructed them.
- a name swapping operation, which is crucial (in combination with name generation) to define the operational semantics of name abstraction matching.

It should be possible to extend MML^N_ν with these features of FreshML. However, one should keep a clear distinction between the types $\langle\mathsf{name}\rangle\tau$ and $[\Sigma|\tau]$. The first type classifies representations of object-level syntax modulo α-conversion, while the second classifies fragments modulo simplification, thus it cannot support program analysis.

MML^N_ν *versus* $\nu^\Box$. 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 $\mathcal{X}_i \subseteq \mathsf{dom}(\Sigma)$.

In $\nu^\Box$ the type of a name X is fixed at name generation time. This is a bad name space management policy, which goes against common practice in programming language design (e.g. of modules systems). MML^N_ν follows the approach of mainstream module languages, where 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.

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 Θ 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 $[\Sigma|\tau]$ corresponds to $\Box_{\mathcal{X}}\tau$ with $\mathcal{X} = \mathsf{dom}(\Sigma)$. Typing rules for $\Box_{\mathcal{X}}\tau$ are related to those for necessity of S4 modal logic, e.g. $\Box_{\mathcal{X}}\tau$ introduction is

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

This rule 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). Without these restrictions substitution would be unsound in the type system of $\nu^\Box$. Such restrictions have no reason to exist in MML^N_ν, because we allow multiple name resolvers, and fragments $\mathsf{b}(r)e$ are formed by abstracting over one name resolver. Furthermore, making name resolvers explicit, avoid the need to introduce *non-standard* forms of substitution.

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

$e \in \nu^{\Box}$	$e' \in \mathsf{MML}^N_\nu$	$e \in \nu^{\Box}$	$e' \in \mathsf{MML}^N_\nu$
x	$\mathsf{ret}\ x$	X	$r.X$
$\lambda x{:}\,\tau.e$	$\mathsf{ret}\ (\lambda x.e')$	$u\langle X_i \to e_i\rangle$	$u\langle r\{X_i{:}\,e'_i\}\rangle$
$e_1\ e_2$	$\mathsf{do}\ x_1 \leftarrow e'_1; x_2 \leftarrow e'_2; x_1 x_2$	$\mathsf{box}\ e$	$\mathsf{ret}\ (\mathsf{b}(r)e')$
$\nu X{:}\,\tau.e$	$\nu X.e'$	$\mathsf{letbox}\ u = e_1\ \mathsf{in}\ e_2$	$\mathsf{do}\ u \leftarrow e'_1; e'_2$

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 quite complex.

In conclusion, the key novelty of MML^N_ν 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).

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.

[AM04] D. Ancona and E. Moggi. A fresh calculus for names management. In Karsai and Visser [KV04].

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

[BCT02] D. Batory, C. Consel, and W. Taha, editors. *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*. Springer, October 2002.

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

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[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.

[3] In [NPar] 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.

[CMT04] C. Calcagno, E. Moggi, and W. Taha. ML-like inference for classifiers. In *Programming Languages & Systems, 13th European Symp. Programming*, volume 2986 of *Lecture Notes in Computer Science*. Springer, 2004.

[CTHL03] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 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.

[ESOP00] *Programming Languages & Systems, 9th European Symp. Programming*, volume 1782 of *Lecture Notes in Computer Science*. Springer, 2000.

[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.

[GS04] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools.* Wiley Publishing Inc, 2004.

[HW03] Christian Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *Programming Languages & Systems, 12th European Symp. Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 284–301. Springer, 2003. Superseded by [HW04].

[HW04] Christian Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. *Sci. Comput. Programming*, 50:189–224, 2004. Supersedes [HW03].

[KV04] G. Karsai and E. Visser, editors. *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*. Springer, October 2004.

[LBCO04] Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors. *Domain-Specific Program Generation.* Number 3016 in Lecture Notes in Computer Science. Springer-Verlag, 2004.

[Met01] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from http://www.cs.rice.edu/˜taha/MetaOCaml/, 2001.

[MF03] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. In *Proc. FoSSaCS '03*, volume 2620 of *Lecture Notes in Computer Science*. Springer, 2003.

[MT00] Elena Machkasova and Franklyn A. Turbak. A calculus for link-time compilation. In ESOP '00 [ESOP00], pages 260–274.

[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.

[NPar] A. Nanevski and F. Pfenning. Staged computations with names and necessity. *J. Funct. Programming*, to appear.

[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 *Lecture Notes in Computer Science*, pages 230–255, Ponte de Lima, Portugal, July 2000. Springer.

[PS03] F. Pfenning and Y. Smaragdakis, editors. *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*. Springer, September 2003.

[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.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd Edition*. Addison Wesley, 2002.

[Tah99] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Inst. of Science and Technology, 1999. Available from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.

[TN03] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.

[TS97] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.

[WV99] J. B. Wells and René Vestergaard. Confluent equational reasoning for linking with first-class primitive modules (long version). A short version is [WV00]. Full paper, 3 appendices of proofs, August 1999.

[WV00] J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In ESOP '00 [ESOP00], pages 412–428. A long version is [WV99].