A Monadic Multi-stage Metalanguage

Eugenio Moggi and Sonia Fagorzi*

DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy

Abstract. We describe a metalanguage MMML, which makes explicit the order of evaluation (in the spirit of monadic metalanguages) and the staging of computations (as in languages for multi-level binding-time analysis). The main contribution of the paper is an operational semantics which is sufficiently detailed for analyzing subtle aspects of multi-stage programming, but also intuitive enough to serve as a reference semantics. For instance, the separation of computational types from code types, makes clear the distinction between a computation for generating code and the generated code, and provides a basis for *multi-lingual* extensions, where a variety of programming languages (aka monads) coexist. The operational semantics consists of two parts: local (semantics preserving) simplification rules, and computation steps executed in a deterministic order (because they may have side-effects). We focus on the computational aspects, thus we adopt a simple type system, that can detect usual type errors, but not the *unresolved link* errors. Because of its explicit annotations, MMML is suitable as an intermediate language.

1 Introduction

Staging a computation into multiple steps is a well-known optimization technique used in algorithms, which exploits information available in early stages for generating code that will be executed in later stages. Multi-stage programming languages, like MetaML (see [9, 15, 16, 3]), provide constructs for expressing staging in a natural and concise manner, and must allow arbitrary interleaving of *code generation* and *computation*. Multi-stage programming is particularly convenient for defining *generative components*, which take as input a specification of user requirements and generate on the fly a customized component, or *mobile applications*, which need to adapt after each move, e.g. by assembling components downloaded remotely to generate code tailored to the local environment.

So far most of the theoretical research on multi-stage programming languages has focused on type systems (for the most recent proposals see [3, 10, 11]). The resulting operational semantics are often instrumental to a particular type system (thus difficult to relate and compare), and often ignore the subtle interactions between code generation and computational effects. In this paper, we provide a deeper understanding of the computational aspects of multi-stage programming, in the framework of a metalanguage with computational types $M\tau$

* Supported by MIUR project NAPOLI and EU project DART IST-2001-33477.

A.D. Gordon (Ed.): FOSSACS 2003, LNCS 2620, pp. 358–374, 2003.

and code types $\langle\tau\rangle$: computational types classify terms describing computations, while code types classify terms representing other terms. We believe that in this framework one can have a fresh look at typing issues, and above all a *generic* approach for adding staging to a programming language (described in a monadic style), including a multi-lingual metalanguage.

An important principle of Haskell [12] is that pure functional evaluation (and all the optimization techniques that come with it) should not be corrupted by the addition of computational effects. In Haskell this separation has been achieved through the use of monads (like monadic IO and monadic state). When describing MMML we adopt this principle not only at the level of types, but also at the level of the operational semantics. In fact, we distinguish between *simplification* (described by local rewrite rules) and *computation* (that may cause side-effects).

Summary. Section 2 describes a general pattern for specifying the operational semantics of monadic metalanguages, which distinguishes simplification from computation. Section 3 exemplifies the general pattern by considering a monadic metalanguage MML for imperative computations. Section 4 introduces an extension MMML with staging, and explains how definitions and results for MML have to be modified and extended. Section 5 gives simple examples of MMML programs, which illustrate the most subtle points of the operational semantics. Section 6 discusses related work and issues specific to MMML.

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

- m, n range over the set N of natural numbers. Furthermore, $m \in \mathsf{N}$ is identified with the set $\{i \in \mathsf{N} | i < m\}$ of its predecessors.
- $\overline{e}$ ranges over the set E^* of finite sequences $(e_i | i \in m)$ of elements of E, and $|\overline{e}|$ denotes its length (i.e. , m). $\overline{e}_1, \overline{e}_2$ denotes the concatenation of $\overline{e}_1$ and $\overline{e}_2$.
- Term equivalence, written $\equiv$, is α-conversion. $\mathrm{FV}(e)$ is the set of variables free in e. If E is a set of terms, then E_0 is the set of $e \in \mathsf{E}$ s.t. $\mathrm{FV}(e) = \emptyset$. $e[x_i := e_i | i \in m]$ (and $e[\overline{x} := \overline{e}]$) denotes parallel substitution (modulo $\equiv$).
- $f : A \stackrel{fin}{\rightarrow} B$ means that f is a partial function from A to B with a finite domain, written $\mathsf{dom}(f)$. We write $\{a_i : b_i | i \in m\}$ for the partial function mapping a_i to b_i (where the a_i must be different, i.e. $a_i = a_j$ implies $i = j$). We use the following operations on partial functions: $\emptyset$ is the everywhere undefined partial function; f_1, f_2 denotes the union of two partial functions with disjoint domains; $f\{a : b\}$ denotes the extension of f to $a \notin \mathsf{dom}(f)$; $f\{a = b\}$ denotes the update of f in $a \in \mathsf{dom}(f)$.
- Given a BNF $e ::= P_1 \mid \ldots \mid P_m$, we write $e += P_{m+1} \mid \ldots \mid P_{m+n}$ as a shorthand for the extended BNF $e ::= P_1 \mid \ldots \mid P_{m+n}$.
- We write $\stackrel{*}{\longrightarrow}$ for the reflexive and transitive closure of a a relation $\longrightarrow$.

2 Monadic Metalanguages, Simplification and Computation

We outline a general pattern for specifying the operational semantics of monadic metalanguages, which distinguishes between *transparent simplification* and *programmable computation*. This is possible because in a monadic metalanguage there is a clear distinction between term-constructors for building terms of computational types, and the other term-constructors that are *computationally irrelevant*. For computationally relevant term-constructors we give an operational semantics that ensures the correct sequencing of computational effects, e.g. by adopting some well-established technique for specifying the operational semantics of programming languages (see [19]), while for computationally irrelevant term-constructors it suffices to give local simplification rules, that can be applied non-deterministically (because they are semantic preserving).

Remark 1. In [18] Wadler adopts a similar style, that distinguishes pure from monadic reduction. However, his pure reduction is a deterministic strategy, while simplification is non-deterministic. In this respect, our approach is related to the Cham [2]: simplification corresponds to heating and computation to reaction.

Combinatory Reduction Systems. We work in the setting of Combinatory Reduction Systems (CRS) [8], which extends Term Rewriting Systems (TRS) with binders. In Section 4 the uniformity of CRS descriptions is exploited for defining the extension with staging *generically* and concisely. In a CRS the syntax of terms is specified by a set C of term-constructors with given arity $\#\colon \mathsf{C} \to \mathsf{N}^*$

$$\boxed{e \in \mathsf{E} ::= x \mid c([\overline{x}_i]e_i | i \in m) \quad \text{with } \#c = (n_i | i \in m) \text{ and } \forall i \in m. |\overline{x}_i| = n_i}$$

Variables x belong to an infinite set X. More complex terms are built by applying a term-constructor c to a sequence of abstractions $[\overline{x}_i]e_i$ binding the free occurrences of the $\overline{x}_i$ in e_i, thus the set of free variables in $c([\overline{x}_i]e_i | i \in m)$ is

$$\mathrm{FV}(c([\overline{x}_i]e_i | i \in m)) \stackrel{\Delta}{=} \cup\{\mathrm{FV}([\overline{x}_i]e_i) | i \in m\} \quad \text{where } \mathrm{FV}([\overline{x}]e) = \mathrm{FV}(e) - \{\overline{x}\}$$

In CRS rewrite rules $e \longrightarrow e'$ can be specified as in TRS, for instance the β-rule is $@(\lambda([x]e'), e) \longrightarrow e'[x := e]$, where e and e' are arbitrary terms. It is possible to give a more schematic syntax for rewrite rules, but it requires metavariables ranging over abstractions.

Given a set T of types τ, a type system deriving judgments of the form $\Gamma \vdash e\colon \tau$, where $\Gamma\colon \mathsf{X} \stackrel{fin}{\to} \mathsf{T}$ is a type assignment, is specified by assigning to each term-constructor c of arity $\#c = (n_i | i \in m)$ a set of type schema $(\overline{\tau}_i \Rightarrow \tau_i | i \in m) \Rightarrow \tau$ consistent with $\#c$, i.e. , $|\overline{\tau}_i| = n_i$ for $i \in m$. More precisely, the typing rules are

$$x \ \frac{}{\Gamma \vdash x\colon \tau}\ \Gamma(x) = \tau \qquad \mathrm{c}\ \frac{\{\Gamma \vdash [\overline{x}_i]e_i\colon \overline{\tau}_i \Rightarrow \tau_i \mid i \in m\}}{\Gamma \vdash c([\overline{x}_i]e_i | i \in m)\colon \tau}\ c\colon (\overline{\tau}_i \Rightarrow \tau_i | i \in m) \Rightarrow \tau$$

where $\Gamma \vdash [\overline{x}]e\colon \overline{\tau} \Rightarrow \tau$ stands for $\Gamma, \{x_i\colon \tau_i | i \in m\} \vdash e\colon \tau$ with $\overline{x} = (x_i | i \in m)$ and $\overline{\tau} = (\tau_i | i \in m)$. Note that $\overline{\tau} \Rightarrow \tau$ is used in type schema, but it is not a type $\tau \in \mathsf{T}$.

Monadic Metalanguages. To specify a monadic metalanguage we define:

- Types $\tau \in \mathsf{T}$, including computational types $M\tau$.
- Terms $e \in \mathsf{E}$, including return $ret(e)$ and monadic do $do(e_1, [x]e_2)$, which corresponds to Haskell do-notation $x \Leftarrow e_1; e_2$.
- A type system, which amounts to give for each term-constructor a set of type schema, in particular for ret and do the type schema are $ret\colon \tau \Rightarrow M\tau$ and $do\colon M\tau_1, (\tau_1 \Rightarrow M\tau_2) \Rightarrow M\tau_2$
- A simplification relation $e \longrightarrow e'$ on terms, namely the *compatible closure* of a set of rewrite rules. By definition of $\longrightarrow$, the induced equivalence is always a congruence. In addition, we require that $\longrightarrow$ satisfies the Church Rosser (CR) and Subject Reduction (SR) properties.
- A *computation* relation $\longmapsto$ on *configurations*. A configuration $Id \in \mathsf{Conf}$ describes the state of a *closed system*, while the relation $\longmapsto$ describes how a closed system may evolve. Usually there is an obvious way to extend $\longrightarrow$ to configurations (preserving the CR property). To formulate a type safety result (along the lines of [19]), we must define well-formed configurations $\vdash Id$, show that both $\longrightarrow$ and $\longmapsto$ preserve well-formedness (for $\longrightarrow$ it should be an easy consequence of SR), and finally establish a progress property for $\Longrightarrow \stackrel{\Delta}{=} \longrightarrow \cup \longmapsto$.

Simplification should be *orthogonal* to computation, i.e. , if $Id_1 \stackrel{*}{\longrightarrow} Id_1'$ and Id_1 can move $Id_1 \longmapsto Id_2$, then Id_1' has a move $Id_1' \longmapsto Id_2'$ s.t. $Id_2 \stackrel{*}{\longrightarrow} Id_2'$.

3 MML: A Monadic Metalanguage for Imperative Computations

We introduce a monadic metalanguage MML for imperative computations, which exemplifies the pattern outlined in Section 2 in a *familiar case*, namely a subset of Haskell with the IO-monad. Moreover, MML provides a starting point for the addition of staging.

- Types $\boxed{\tau \in \mathsf{T} ::= \mathsf{nat} \mid M\tau \mid \tau_1 \to \tau_2 \mid \mathsf{ref}\ \tau}$. The type nat of natural numbers avoids a degenerate BNF (we will ignore it most of the time).
- Term-constructors $\boxed{c \in \mathsf{C} ::= ret \mid do \mid \lambda \mid @ \mid new \mid get \mid set \mid l}$. Locations l belong to an infinite set L (they are not allowed in user-written programs, but are instrumental to the operational semantics). The type schema for term-constructors (from which one can infer also their arity) are
  - $ret\colon \tau \Rightarrow M\tau$ and $do\colon M\tau_1, (\tau_1 \Rightarrow M\tau_2) \Rightarrow M\tau_2$
  - $@\colon (\tau_1 \to \tau_2), \tau_1 \Rightarrow \tau_2$ and $\lambda\colon (\tau_1 \Rightarrow \tau_2) \Rightarrow (\tau_1 \to \tau_2)$
  - $new\colon \tau \Rightarrow M(\mathsf{ref}\ \tau)$, $get\colon \mathsf{ref}\ \tau \Rightarrow M\tau$ and $set\colon \mathsf{ref}\ \tau, \tau \Rightarrow M(\mathsf{ref}\ \tau)$

  a signature $\Sigma\colon \mathsf{L} \stackrel{fin}{\to} \mathsf{T}$ gives the type to locations, i.e. , $l\colon \mathsf{ref}\ \tau$ when $\Sigma(l) = \tau$.

– The BNF for terms $e \in \mathsf{E}$ generated by the term-constructors above is
$\boxed{e ::= x \mid ret(e) \mid do(e_1, [x]e_2) \mid \lambda([x]e) \mid @(e_1, e_2) \mid new(e) \mid get(e) \mid set(e_1, e_2) \mid l}$
$\lambda([x]e)$ and $@(e_1, e_2)$ are λ-abstraction $\lambda x.e$ and application $e_1 e_2$; *new*, *get* and *set* are the ML-like operations $\mathsf{ref}\ e$, $!e$ and $e_1 := e_2$ on references.

The type system is parametric in Σ, and the rules for deriving judgments of the form $\Gamma \vdash_\Sigma e : \tau$ are

$$x\ \frac{}{\Gamma \vdash_\Sigma x : \tau}\ \Gamma(x) = \tau \qquad l\ \frac{}{\Gamma \vdash_\Sigma l : \mathsf{ref}\ \tau}\ \Sigma(l) = \tau$$

$$\mathrm{c}\ \frac{\{\Gamma \vdash_\Sigma [\overline{x}_i]e_i : \overline{\tau}_i \Rightarrow \tau_i \mid i \in m\}}{\Gamma \vdash_\Sigma c([\overline{x}_i]e_i \mid i \in m) : \tau}\ c : (\overline{\tau}_i \Rightarrow \tau_i \mid i \in m) \Rightarrow \tau$$

Simplification $\longrightarrow$ is the compatible closure of $@(\lambda([x]e_2), e_1) \longrightarrow e_2[x := e_1]$, i.e. , β-reduction. We write $=$ for β-equivalence, i.e. , the reflexive, symmetric and transitive closure of $\longrightarrow$. We recall the properties of simplification (β-reduction) relevant for our purposes.

Proposition 1 (Congr). *The equivalence $=$ induced by $\longrightarrow$ is a congruence.*

Proposition 2 (CR). *The simplification relation $\longrightarrow$ is confluent.*

Proposition 3 (SR). *If $\Gamma \vdash_\Sigma e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash_\Sigma e' : \tau$.*

Remark 2. Several extensions can be handled at the level of simplification.

– The extension with a datatype, like nat or $\tau_1 \times \tau_2$, amounts to add term-constructors for introduction and elimination ($zero : \mathsf{nat}$, $succ : \mathsf{nat} \Rightarrow \mathsf{nat}$ and $case : \mathsf{nat}, \tau, (\mathsf{nat} \Rightarrow \tau) \Rightarrow \tau$) and simplification rules describing how they interact ($case(zero, e_0, [x]e_1) \longrightarrow e_0$ and $case(succ(e), e_0, [x]e_1) \longrightarrow e_1[x := e]$).
– Recursive definitions can be handled by a term-constructor $\mathsf{fix} : (\tau \Rightarrow \tau) \Rightarrow \tau$ with simplification rule $\mathsf{fix}([x]e) \longrightarrow e[x := \mathsf{fix}([x]e)]$. However, if one wants simplification of well-typed terms to terminate, then the type schema for fix should be $(M\tau \Rightarrow M\tau) \Rightarrow M\tau$ and $\mathsf{fix}([x]e)$ becomes a *computation redex*.
– A test for equality of references $\mathsf{ifeq} : \mathsf{ref}\ \tau, \mathsf{ref}\ \tau, \tau', \tau' \Rightarrow \tau'$ with simplification rules $\mathsf{ifeq}(l, l, e_1, e_2) \longrightarrow e_1$ and $\mathsf{ifeq}(l_1, l_2, e_1, e_2) \longrightarrow e_2$ when $l_1 \neq l_2$.

3.1 Computation

We define configurations $Id \in \mathsf{Conf}$ (and the auxiliary notions of store, evaluation context and computation redex) and the computation relation $Id \longmapsto Id' \mid ok$ (see Table 1).

– Stores $\mu \in \mathsf{S} \stackrel{\Delta}{=} \mathsf{L} \stackrel{fin}{\rightarrow} \mathsf{E}$ map locations to their contents.
– Evaluation contexts $\boxed{E \in \mathsf{EC} ::= \Box \mid E[do(\Box, [x]e)]}$.

- Configurations $(\mu, e, E) \in \mathsf{Conf} \stackrel{\Delta}{=} \mathsf{S} \times \mathsf{E} \times \mathsf{EC}$ consist of the current store μ, the program fragment e under consideration and its evaluation context E.
- Computation redexes $\boxed{r \in \mathsf{R} ::= do(e_1, [x]e_2) \mid ret(e) \mid new(e) \mid get(l) \mid set(l, e)}$.

When the program fragment under consideration is a computation redex, it enables a computation step with no need for further simplification (see Theorem 1).

Administrative steps, involve only the evaluation context

A.0 $(\mu, ret(e), \Box) \longmapsto ok$
A.1 $(\mu, ret(e_1), E[do(\Box, [x]e_2)]) \longmapsto (\mu, e_2[x := e_1], E)$
A.2 $(\mu, do(e_1, [x]e_2), E) \longmapsto (\mu, e_1, E[do(\Box, [x]e_2)])$

Imperative steps, involve only the store

I.1 $(\mu, new(e), E) \longmapsto (\mu\{l : e\}, ret(l), E)$, where $l \notin \mathsf{dom}(\mu)$
I.2 $(\mu, get(l), E) \longmapsto (\mu, ret(e), E)$, provided $e = \mu(l)$
I.3 $(\mu, set(l, e), E) \longmapsto (\mu\{l = e\}, ret(l), E)$, provided $l \in \mathsf{dom}(\mu)$

Table 1. Computation Relation for MML

The confluent simplification relation $\longrightarrow$ on terms extends in the obvious way to a confluent relation (denoted $\longrightarrow$) on stores, evaluation contexts, computation redexes and configurations.

A *complete program* corresponds to a closed term $e \in \mathsf{E}_0$ (with no occurrences of locations l), and its evaluation starts from the *initial configuration* $(\emptyset, e, \Box)$. The following properties ensure that only closed configurations are reachable (by $\longrightarrow$ and $\longmapsto$ steps) from initial ones.

Lemma 1.

1. *If* $(\mu, e, E) \longrightarrow (\mu', e', E')$, *then* $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$ *and* $\mathrm{FV}(\mu') \subseteq \mathrm{FV}(\mu)$, $\mathrm{FV}(e') \subseteq \mathrm{FV}(e)$ *and* $\mathrm{FV}(E') \subseteq \mathrm{FV}(E)$.
2. *If* $Id \longmapsto Id'$ *and* Id *is closed, then* Id' *is closed.*

When the program fragment under consideration is a computation redex, it does not matter whether simplification is done before or after computation.

Theorem 1 (Bisim). *If* $Id \equiv (\mu, e, E)$ *with* $e \in \mathsf{R}$ *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* $\mathsf{Conf} \cup \{ok\}$.

Proof. An equivalent statement, but easier to prove, is obtained by replacing $\xrightarrow{*}$ with one-step parallel reduction. A key observation for proving the bisimulation result is that simplification applied to a computation redex r and an evaluation context E does not change the relevant structure (of r and E) for determining the computation step among those in Table 1.

3.2 Type Safety

We go through the proof of type safety. The result is standard and unsurprising, but we make some adjustments to the Subject Reduction (SR) and Progress properties, in order to stress the role of simplification $\longrightarrow$ and computation $\longmapsto$, when they are not bundled in one deterministic reduction strategy on configurations. First of all, we define well-formedness for configurations $\vdash_\Sigma Id$ and evaluation contexts $\Box\colon M\tau \vdash_\Sigma E\colon M\tau'$.

Definition 1. *We write* $\vdash_\Sigma (\mu, e, E) \overset{\Delta}{\Longleftrightarrow}$

- $\mathsf{dom}(\Sigma) = \mathsf{dom}(\mu)$
- $\mu(l) = e_l$ *and* $\Sigma(l) = \tau_l$ *imply* $\vdash_\Sigma e_l\colon \tau_l$
- *there exists* τ *such that* $\vdash_\Sigma e\colon M\tau$ *is derivable*
- *there exists* τ' *such that* $\Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ *is derivable (see Table 2)*

$$\Box\ \frac{}{\Box\colon M\tau' \vdash_\Sigma \Box\colon M\tau'} \qquad do\ \frac{\Box\colon M\tau_2 \vdash_\Sigma E\colon M\tau' \quad \vdash_\Sigma [x]e\colon \tau_1 \Rightarrow M\tau_2}{\Box\colon M\tau_1 \vdash_\Sigma E[do(\Box, [x]e)]\colon M\tau'}$$

Table 2. Well-formed Evaluation Contexts for MML

Theorem 2 (SR).

1. *If* $\vdash_\Sigma Id_1$ *and* $Id_1 \longrightarrow Id_2$, *then* $\vdash_\Sigma Id_2$
2. *If* $\vdash_{\Sigma_1} Id_1$ *and* $Id_1 \longmapsto Id_2$, *then exists* $\Sigma_2 \supseteq \Sigma_1$ *s.t.* $\vdash_{\Sigma_2} Id_2$

Proof. The first claim is an easy consequence of Proposition 3. The second is proved by case-analysis on the computation rules of Table 1.

Theorem 3 (Progress). *If* $\vdash_\Sigma (\mu, e, E)$, *then one of the following holds*

1. $e \notin \mathsf{R}$ *and* $e \longrightarrow$, *or*
2. $e \in \mathsf{R}$ *and* $(\mu, e, E) \longmapsto$

Proof. When $e \in \mathsf{R}$ we have $(\mu, e, E) \longmapsto$, e.g. when e is $get(l)$ or $set(l, e')$, then $l \in \mathsf{dom}(\mu)$ by well-formedness of the configuration. When $e \notin \mathsf{R}$, then e cannot be a $\longrightarrow$-normal form, otherwise we get a contradiction with $\vdash_\Sigma e\colon M\tau$.

4 MMML: A Multi-stage Extension of MML

We describe a monadic metalanguage MMML obtained by adding staging to MML. At the level of syntax, type system and simplification the extension is *generic*, i.e. , applicable to any monadic metalanguage (as defined in Section 2).

- The BNF for types $\boxed{\tau \in \mathsf{T}+ = \langle\tau\rangle}$ is extended with code types.
- The BNF for term-constructors $\boxed{c \in \mathsf{C}+ = up \mid dn \mid c_V \mid c_M}$ is extended with *up*, *dn* and two recursive productions c_V and c_M, which capture the reflective nature of the extension (the set of term-constructors for MMML is infinite, although that for MML is finite). The type schema for the additional term-constructors are
  - $up\colon \tau \Rightarrow \langle\tau\rangle$ is MetaML cross-stage persistence (aka binary inclusion).
  - $dn\colon \langle\tau\rangle \Rightarrow M\tau$ is compilation of (potentially open) code. An attempt to compile open code causes an *unresolved link* error (an effect not present in MML), thus *dn* has a computational result type.
  - if $c\colon (\overline{\tau}_i \Rightarrow \tau_i | i \in m) \Rightarrow \tau$, then
    - $c_V\colon (\langle\overline{\tau}_i\rangle \Rightarrow \langle\tau_i\rangle | i \in m) \Rightarrow \langle\tau\rangle$ builds code representing a term $c(\ldots)$
    - $c_M\colon (\langle\overline{\tau}_i\rangle \Rightarrow M\langle\tau_i\rangle | i \in m) \Rightarrow M\langle\tau\rangle$ builds a computations that generates code representing $c(\ldots)$

    where $\langle\tau_i | i \in m\rangle$ stands for the sequence $(\langle\tau_i\rangle | i \in m)$. For instance, $\lambda_V\colon (\langle\tau_1\rangle \Rightarrow \langle\tau_2\rangle) \Rightarrow \langle\tau_1 \to \tau_2\rangle$ and $\lambda_M\colon (\langle\tau_1\rangle \Rightarrow M\langle\tau_2\rangle) \Rightarrow M\langle\tau_1 \to \tau_2\rangle$.

  The key difference between c_V and c_M (reflected in their type schema) is that generating code with c_M may have computational effects, while with c_V does not. For instance, the computation $\lambda_M([x]e)$ *generates a fresh name* (a new effect related to computation under a binder), performs the computation e to generate the code e' for the body of the λ-abstraction, and finally returns the code $\lambda_V([x]e')$ for the λ-abstraction.

The BNF for terms $e \in \mathsf{E}$ and the type system (for deriving judgments of the form $\Gamma \vdash_\Sigma e\colon \tau$) are extended in the only possible way, given the type schema for the term-constructors. In MMML (unlike $\lambda^{\bigcirc}$ and MetaML) there is no need to include level information in typing judgments, since it is already explicit in types and terms. For instance, a MetaML type τ at level 1 becomes $\langle\tau\rangle$ in MMML, and a λ at level 1 becomes a λ_V or λ_M.

Simplification $\longrightarrow$ is unchanged, i.e. , no new simplification rules are added. The properties of simplification established in Section 3 (i.e. , Proposition 1, 2 and 3) continue to hold and their proofs are unchanged.

Remark 3. One may wonder whether there is a need to have both c_M and c_V, or whether c_M can be defined in terms of c_V and term-constructors for computational types. Indeed, this is the case when c is **not a binder**. For instance, when $c\colon \tau_1 \Rightarrow \tau_2$ and $e\colon M\tau_1$, we could define $c_M(e)$ as $do(e, [x]ret(c_V(x)))$.

However, when c is **a binder**, like λ, one cannot move the computation of the body of $\lambda_M([x]e)$ outside the binder. One could adopt a more concrete representation of terms using first-order abstract syntax (FOAS), and introduce a

monadic operation *gensym*: $M\langle\tau\rangle$ to generate a fresh name (see [6]). But in this approach λ_V is no longer a binder, and this would be a drastic loss of abstraction for a reference semantics.

In λ-calculus one can encode a term-constructor c as a constant c' of higher-order type. For instance, $do(e_1, [x]e_2)$ becomes $do'@e_1@(\lambda x.e_2)$, where we adopt the standard infix notation for application @. Then we can use c'_V to encode c_M and c_V. For instance, $do_V(e_1, [x]e_2)$ becomes $do'_V@_V e_1@_V(\lambda_V x.e_2)$, and $do_M(e_1, [x]e_2)$ becomes $do@e_1@(\lambda c.do@(\lambda_M x.e_2)@(\lambda f.do'_V@_V c@_V f))$. With this encoding (unlike FOAS) there is no loss of abstraction, moreover it gives better control on code generation, e.g. $do_M(e_1, [x]e_2)$ (and its encoding) computes e_1 first, while $do@(\lambda_M x.e_2)@(\lambda f.do@e_1@(\lambda c.do'_V@_V c@_V f))$ computes e_2 first.

4.1 Computation

We now define configurations and the computation relation $Id \longmapsto Id' \mid ok \mid \mathsf{err}$ for MMML (see Table 3), where err indicates an unresolved link error at run-time. We must account for run-time errors, because we have adopted a permissive (and simple) type system. In the following we stress what auxiliary notions need to be changed when going from MML to MMML.

Remark 4. When adding staging, the modifications to the definition of $\longmapsto$ are fairly *modular*, but we cannot rely on a general theory (like rewrite rules for CRS) as in the case of simplification.

- Stores $\mu \in \mathsf{S} \stackrel{\Delta}{=} \mathsf{L} \stackrel{fin}{\to} \mathsf{E}$ are unchanged.
- Evaluation contexts $\boxed{E \in \mathsf{EC}+ = E[c_M(\overline{v}, [\overline{x}]\Box, \overline{f})]}$ are extended with one production, where $c \in \mathsf{C}$, $f ::= [\overline{x}]e$ is an abstraction, $v ::= [\overline{x}]ret(e)$ is a value abstraction. Moreover, $\overline{v}$, $\overline{x}$ and $\overline{f}$ must be consistent with the arity of c, for instance $E[\lambda_M([x]\Box)]$, $E[do_M(\Box, [x]e)]$ and $E[do_M(ret(e), [x]\Box)]$. Intuitively $E[\lambda_M([x]\Box)]$ says that the program fragment under consideration is generating code for the body of a λ-abstraction.
- A configuration $(X|\mu, e, E) \in \mathsf{Conf} \stackrel{\Delta}{=} \mathcal{P}_{fin}(\mathsf{X}) \times \mathsf{S} \times \mathsf{E} \times \mathsf{EC}$ has an additional component, i.e. , the set X of *names* generated so far. A name may leak outside the scope of its binder, thus X grows as the computation progresses.
- Computation redexes $\boxed{r \in \mathsf{R} += c_M(\overline{f}) \mid dn(vc)}$ are extended with two productions, where $c \in \mathsf{C}$, $\overline{f}$ must be consistent with the arity of c, and $\boxed{vc \in \mathsf{VC} ::= x \mid up(e) \mid c_V([\overline{x}_i]vc_i | i \in m)}$ is a code value. The redex $c_M(\overline{f})$ may generate fresh names, while $dn(vc)$ may cause an unresolved link error.

Compilation dn takes a code value vc of type $\langle\tau\rangle$ and computes the term e of type τ represented by vc (or fails if e does not exist). The represented term e is given by an operation similar to MetaML's demotion.

Definition 2 (Demotion). *The partial function* $\_\downarrow$ *mapping* $vc \in \mathsf{VC}$ *to the represented term is given by*

– $x\!\downarrow$ *is undefined;* $up(e)\!\downarrow = e$ *(this is a base case, like* x*);*
$c_V([\overline{x}_i]vc_i|i \in m)\!\downarrow = c([\overline{x}_i]e_i|i \in m)$ *when* $e_i = vc_i[\overline{x}_i := up(\overline{x}_i)]\!\downarrow$ *for* $i \in m$

where $up(\overline{x})$ *is the sequence* $(up(x_i)|i \in m)$ *when* $\overline{x} = (x_i|i \in m)$

Administrative and Imperative steps are as in Table 1, they do not modify the set X. Code generation steps, involve only the set X and the evaluation context

G.0 $(X|\mu, c_M, E) \longmapsto (X|\mu, ret(c_V), E)$ when the arity of c is ()
G.1 $(X|\mu, c_M([\overline{x}]e, \overline{f}), E) \longmapsto (X, \overline{x}|\mu, e, E[c_M([\overline{x}]\Box, \overline{f})])$ with $\overline{x}$ **renamed to avoid clashes** with X. In particular $(X|\mu, \lambda_M([x]e), E) \longmapsto (X, x|\mu, e, E[\lambda_M([x]\Box)])$
G.2 $(X|\mu, ret(e), E[c_M(\overline{v}, [\overline{x}]\Box)]) \longmapsto (X|\mu, ret(c_V(\overline{f}, [\overline{x}]e)), E)$ where $\overline{v} = ([\overline{x}_i]ret(e_i)|i \in m)$ and $\overline{f} = ([\overline{x}_i]e_i|i \in m)$. The free occurrences of $\overline{x}$ in e **get captured** by c_V, e.g. $(X|\mu, ret(e), E[\lambda_M([x]\Box)]) \longmapsto (X|\mu, ret(\lambda_V([x]e)), E)$
G.3 $(X|\mu, ret(e_1), E[c_M(\overline{v}, [\overline{x}_1]\Box, [\overline{x}_2]e_2, \overline{f})]) \longmapsto$
$(X, \overline{x}_2|\mu, e_2, E[c_M(\overline{v}, [\overline{x}_1]ret(e_1), [\overline{x}_2]\Box, \overline{f})])$ with $\overline{x}_2$ **renamed to avoid clashes** with X, and the free occurrences of $\overline{x}_1$ in e_1 **captured** by c_M.

Compilation step, may cause a run-time error

C.1 $(X|\mu, dn(vc), E) \longmapsto \begin{cases} (X|\mu, ret(e), E) & \text{if } e = vc\!\downarrow \\ \mathsf{err} & \text{if } vc\!\downarrow \text{ undefined} \end{cases}$

Table 3. Computation Relation for MMML

In an evaluation context for MMML, e.g. $E[\lambda_M([x]\Box)]$, the hole $\Box$ can be within the scope of a binder, thus an evaluation context E has not only a set of free variables, but also a sequence of captured variables.

Definition 3. *The sequence* $\mathrm{CV}(E)$ *of* captured variables *and the set* $\mathrm{FV}(E)$ *of* free variables *are defined by induction on the structure of* E

– $\mathrm{CV}(\Box) \triangleq \emptyset$ $\quad \mathrm{CV}(E[do(\Box, [x]e)]) \triangleq \mathrm{CV}(E)$
$\mathrm{CV}(E[c_M(\overline{v}, [\overline{x}]\Box, \overline{f})]) \triangleq \mathrm{CV}(E), \overline{x}$ *in particular* $\mathrm{CV}(E[\lambda_M([x]\Box)]) \triangleq \mathrm{CV}(E), x$
– $\mathrm{FV}(\Box) \triangleq \emptyset$ $\quad \mathrm{FV}(E[do(\Box, [x]e)]) \triangleq \mathrm{FV}(E) \cup (\mathrm{FV}([x]e) - \mathrm{CV}(E))$
$\mathrm{FV}(E[c_M(\overline{v}, [\overline{x}]\Box, \overline{f})]) \triangleq \mathrm{FV}(E) \cup (\mathrm{FV}(\overline{v}, \overline{f}) - \mathrm{CV}(E))$

As in the case of MML, the confluent simplification relation on terms extends to a confluent relation on the other syntactic categories. Also for MMML we can prove that only closed configurations are reachable from an initial one $(\emptyset|\emptyset, e, \Box)$, where $e \in \mathsf{E}_0$. However, the second clause of Lemma 2 is more subtle, in particular it ensures that $\mathrm{FV}(E)$ and $\mathrm{CV}(E)$ remain disjoint.

Lemma 2.

1. *If* $(X|\mu, e, E) \longrightarrow (X'|\mu', e', E')$, *then* $X' = X$, $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$, $\mathrm{CV}(E') = \mathrm{CV}(E)$, $\mathrm{FV}(\mu') \subseteq \mathrm{FV}(\mu)$, $\mathrm{FV}(e') \subseteq \mathrm{FV}(e)$ *and* $\mathrm{FV}(E') \subseteq \mathrm{FV}(E)$.

2. *If* $(X|\mu, e, E) \longmapsto\!\!\!\!\rightarrow (X'|\mu', e', E')$, $\mathrm{FV}(\mu, e) \cup \mathrm{CV}(E) \subseteq X$ *and* $\mathrm{FV}(E) \subseteq X - \mathrm{CV}(E)$, *then* $X \subseteq X'$, $dom(\mu) \subseteq dom(\mu')$, $\mathrm{FV}(\mu', e') \cup \mathrm{CV}(E') \subseteq X'$ *and* $\mathrm{FV}(E') \subseteq X' - \mathrm{CV}(E')$.

The bisimulation result (Theorem 1) is basically unchanged, but the proof must cover additional cases corresponding to the computation rules in Table 3.

4.2 Type Safety

In MMML the definitions of well-formed configuration $\Delta \vdash_\Sigma Id$ and evaluation context $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ must take into account the set X. For this reason we need a type assignment Δ which maps names $x \in X$ to code types $\langle\tau\rangle$.

Definition 4. *We write* $\Delta \vdash_\Sigma (X|\mu, e, E) \overset{\Delta}{\Longleftrightarrow}$

- $dom(\Sigma) = dom(\mu)$ *and* $dom(\Delta) = X$
- $\mu(l) = e_l$ *and* $\Sigma(l) = \tau_l$ *imply* $\Delta \vdash_\Sigma e_l\colon \tau_l$
- *there exists* τ *such that* $\Delta \vdash_\Sigma e\colon M\tau$ *is derivable*
- *there exists* τ' *such that* $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ *is derivable (see Table 4).*

$$\Box\ \frac{}{\Delta, \Box\colon M\tau' \vdash_\Sigma \Box\colon M\tau'} \qquad do\ \frac{\Delta, \Box\colon M\tau_2 \vdash_\Sigma E\colon M\tau' \quad \Delta \vdash_\Sigma [x]e\colon \tau_1 \Rightarrow M\tau_2}{\Delta, \Box\colon M\tau_1 \vdash_\Sigma E[do(\Box, [x]e)]\colon M\tau'}$$

$$c_M\ \frac{\begin{array}{c}\Delta, \Box\colon M\langle\tau\rangle \vdash_\Sigma E\colon M\tau' \\ \{\Delta \vdash_\Sigma v_i\colon \langle\overline{\tau}_i\rangle \Rightarrow M\langle\tau_i\rangle \mid i \in m\} \quad \{\Delta \vdash_\Sigma f_i\colon \langle\overline{\tau}_{m+1+i}\rangle \Rightarrow M\langle\tau_{m+1+i}\rangle \mid i \in n\}\end{array}}{\Delta, \{x_k\colon \langle\tau'_k\rangle | k \in p\}, \Box\colon M\langle\tau_m\rangle \vdash_\Sigma E[c_M(\overline{v}, [\overline{x}]\Box, \overline{f})]\colon M\tau'}\ \text{cond}$$

where the side-condition (cond) is: $\overline{v} = (v_i | i \in m)$ and $\overline{f} = (f_i | i \in n)$; $c_M\colon (\langle\overline{\tau}_i\rangle \Rightarrow M\langle\tau_i\rangle | i \in m + 1 + n) \Rightarrow M\langle\tau\rangle$; $\overline{\tau}_m = (\tau'_k | k \in p)$ and $\overline{x} = (x_k | k \in p)$

$$\text{in particular}\ \ \lambda_M\ \frac{\Delta, \Box\colon M\langle\tau_1 \to \tau_2\rangle \vdash_\Sigma E\colon M\tau'}{\Delta, x\colon \langle\tau_1\rangle, \Box\colon M\langle\tau_2\rangle \vdash_\Sigma E[\lambda_M([x]\Box)]\colon M\tau'}$$

Table 4. Well-formed Evaluation Contexts for MMML

Remark 5. The formation rule (c_M) for an evaluation context $E[c_M(\overline{v}, [\overline{x}]\Box, \overline{f})]$ says that the captured variables $\overline{x}$ must have a code type (this is consistent with the code generation rules (G.1) and (G.3) of Table 3) and that they should not occur free in E, $\overline{v}$ or $\overline{f}$ (this is consistent with the second property in Lemma 2).

Lemma 3. *If* $\Gamma \vdash_\Sigma vc\colon \langle\tau\rangle$ *and* $e = vc\downarrow$, *then* $\Gamma \vdash_\Sigma e\colon \tau$.

We can now formulate the SR and progress properties for MMML.

Theorem 4 (SR).

1. *If* $\Delta \vdash_\Sigma Id_1$ *and* $Id_1 \longrightarrow Id_2$*, then* $\Delta \vdash_\Sigma Id_2$
2. *If* $\Delta_1 \vdash_{\Sigma_1} Id_1$ *and* $Id_1 \longmapsto Id_2$*, then exist* $\Sigma_2 \supseteq \Sigma_1$ *and* $\Delta_2 \supseteq \Delta_1$ *s.t.* $\Delta_2 \vdash_{\Sigma_2} Id_2$

Proof. The first claim is straightforward (see Theorem 2). The second is proved by case-analysis on the computation rules, so we must cover the additional cases for the computation rules in Table 3, e.g.

(G.1) if Id_1 is $(X|\mu, \lambda_M([x]e), E)$, then Id_2 is $(X, x|\mu, e, E[\lambda_M([x]\Box)])$ and the typings $\Delta_1, x\colon\langle\tau_1\rangle \vdash_{\Sigma_1} e\colon M\langle\tau_2\rangle$ and $\Delta_1, \Box\colon M\langle\tau_1 \to \tau_2\rangle \vdash_{\Sigma_1} E\colon\tau'$ are derivable. Therefore we can take $\Sigma_2 \equiv \Sigma_1$ and $\Delta_2 \equiv \Delta_1, x\colon\langle\tau_1\rangle$.

(C.1) if Id_1 is $(X|\mu, dn(vc), E)$, then Id_2 is $(X, x|\mu, ret(e), E)$ with $e = vc\downarrow$ and the typings $\Delta_1 \vdash_{\Sigma_1} vc\colon\langle\tau\rangle$ and $\Delta_1, \Box\colon M\tau \vdash_{\Sigma_1} E\colon\tau'$ are derivable. By Lemma 3 $\Delta_1 \vdash_{\Sigma_1} e\colon\tau$ is derivable, therefore we can take $\Sigma_2 \equiv \Sigma_1$ and $\Delta_2 \equiv \Delta_1$.

Lemma 4. *If* $\Delta \vdash_\Sigma e\colon\tau$ *and* e *is a* $\longrightarrow$*-normal form, then*

- $\tau \equiv \mathsf{nat}$ *implies* e *is a natural number*
- $\tau \equiv M\tau$ *implies* e *is a computation redex*
- $\tau \equiv \mathsf{ref}\ \tau$ *implies* e *is a location*
- $\tau \equiv \langle\tau'\rangle$ *implies* e *is a code value*
- $\tau \equiv (\tau_1 \to \tau_2)$ *implies* e *is a* λ*-abstraction*

Proof. By induction on the derivation of $\Delta \vdash_\Sigma e\colon\tau$. The base cases are: x, *up*, l, λ, *ret*, *do*, *new* and c_M. The inductive steps are: *get*, *set*, *dn*, c_V and @ (@ is impossible because by the IH one would have a β-redex).

Theorem 5 (Progress). *If* $\Delta \vdash_\Sigma (X|\mu, e, E)$*, then one of the following holds*

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

Proof. When $e \in \mathsf{R}$ we have $(\mu, e, E) \longmapsto$ (see Theorem 3). When $e \notin \mathsf{R}$, then e cannot be a $\longrightarrow$-normal form. otherwise we get a contradiction with $\Delta \vdash_\Sigma e\colon M\tau$ and Lemma 4.

5 Examples

We give simple examples of computations in MMML to illustrate subtle points of multi-stage programming. For readability, we use Haskell's do-notation $x \Leftarrow e_1; e_2$ (or $e_1; e_2$) for $do(e_1, [x]e_2)$ (when $x \notin \mathrm{FV}(e_2)$) and write $\lambda_B x.e$ for $\lambda_B([x]e)$.

Scope extrusion: a bound variable x leaks in the store.

$$l \Leftarrow new(0_V); (\lambda_M x.set(l,x); ret(x)) : M\langle \mathsf{nat} \to \mathsf{nat} \rangle$$

1. $(\emptyset \mid \emptyset, \quad new(0_V), \quad l \Leftarrow \Box; \lambda_M x.set(l,x); ret(x))$ create a location l
2. $(\emptyset \mid l = 0_V, \quad \lambda_M x.set(l,x); ret(x), \quad \Box)$ generate a fresh name x
3. $(x \mid l = 0_V, \quad set(l,x), \quad \lambda_M x.\Box; ret(x))$ assign x to l
4. $(x \mid l = x, \quad ret(x), \quad \lambda_M x.\Box)$ complete code generation of λ-abstraction
5. $(x \mid l = x, \quad ret(\lambda_V x.x), \quad \Box)$ x is bound by λ_V, but a copy is also left in the store.

The semantics in [3] is more conservative, when a variable leaks in the store it is bound by *dead-code annotation.*, but on closed values the two semantics *agree.*

Recapturing of extruded variable by its binder.

$$\lambda_M x.\ l \Leftarrow new(x); get(l) : M\langle \tau \to \tau \rangle$$

1. $(\emptyset \mid \emptyset, \quad \lambda_M x.\ l \Leftarrow new(x); get(l), \quad \Box)$ generate x, then create l
2. $(x \mid l = x, \quad get(l), \quad \lambda_M x.\Box)$ get content of l
3. $(x \mid l = x, \quad ret(x), \quad \lambda_M x.\Box)$ complete code generation of λ-abstraction
4. $(x \mid l = x, \quad ret(\lambda_V x.x), \quad \Box)$ x is bound by λ_V.

This form of recapturing is allowed by [17], but not by [3].

No recapturing of extruded variable by another binder using the *same* name.

$$l \Leftarrow new(0_V); (\lambda_M x.set(l,x); ret(x)); z \Leftarrow get(l); ret(\lambda_V x.z) : M\langle \tau \to \mathsf{nat} \rangle$$

1. $(\emptyset \mid l = 0_V, \quad (\lambda_M x.set(l,x); ret(x)), \quad \Box; z \Leftarrow get(l); ret(\lambda_V x.z))$
   generate x and assign it to l
2. $(x \mid l = x, \quad ret(\lambda_V x.x), \quad \Box; z \Leftarrow get(l); ret(\lambda_V x.z))$
   first code generation completed
3. $(x \mid l = x, \quad get(l), \quad z \Leftarrow \Box; ret(\lambda_V x.z))$ get content of l
4. $(x \mid l = x, \quad ret(x), \quad z \Leftarrow \Box; ret(\lambda_V x.z))$
   complete code generation of λ-abstraction
5. $(x \mid l = x, \quad ret(\lambda_V x'.x), \quad \Box)$
   the bound variable x is renamed by substitution $ret(\lambda_V x.z)[z := x]$

No recapturing of extruded variable by its binder after code generation.

$$\begin{aligned} l \Leftarrow new(0_V); z \Leftarrow (\lambda_M x.\lambda_M y.set(l,y); ret(x)); \\ f \Leftarrow dn(z); u \Leftarrow get(l); ret(f\ u) : M(\mathsf{nat} \to \langle \mathsf{nat} \rangle) \end{aligned}$$

1. $(x, y \mid l = y, \quad ret(\lambda_V x.\lambda_V y.x), \quad z \Leftarrow \Box; f \Leftarrow dn(z); u \Leftarrow get(l); ret(f\ u))$
   code generation completed, y is bound by λ_V and leaked in the store
2. $(x, y \mid l = y, \quad dn(\lambda_V x.\lambda_V y.x), \quad f \Leftarrow \Box; u \Leftarrow get(l); ret(f\ u))$ compile code

3. $(x, y \mid l = y, \quad ret(\lambda x.\lambda y.x), \quad f \Leftarrow \Box; u \Leftarrow get(l); ret(f\ u))$
   get content of l and apply f to it
4. $(x, y \mid l = y, \quad ret((\lambda x.\lambda y.x)\ y), \Box)$ the result simplifies to $(\lambda y'.y)$, because the bound variable y is renamed by β-reduction.

When y is recaptured by λ_V, it becomes a bound variable and can be renamed. Therefore, the connection with the (free) occurrences of y left in the store (or the program fragment under consideration) is lost.

6 Related Work and Discussion

We discuss related work and some issues specific to MMML. A more general discussion of open issues in meta-programming can be found in [14].

Comparison with MetaML, $\lambda^{\bigcirc}$ *and* $\lambda^{\mathfrak{M}}$. The motivation for looking at the interactions between computational effects and run-time code generation comes from MetaML [9, 15, 16, 3]. We borrow code types from MetaML (and $\lambda^{\bigcirc}$ of [4]), but use annotated term-constructors as in $\lambda^{\mathfrak{M}}$ of [4] (see also [7]), so that simplification and computation rules are *level insensitive*. Indeed, the term-constructors $c \in \mathsf{C}$ of MMML can be given by an alternative BNF

$$c ::= ret_B \mid do_B \mid \lambda_B \mid @_B \mid new_B \mid get_B \mid set_B \mid l_B \mid up_B \mid dn_B \quad \text{with } B \in \{V, M\}^*$$

For instance, λ_B is λ when B is empty; if c is λ_B, then c_V and c_M are given by λ_{BV} and λ_{BM} respectively. However, MMML's annotations are sequences $B \in \{V, M\}^*$, while those of $\lambda^{\mathfrak{M}}$ are natural number n. A sequence B identifies a natural number n, namely the length of B, moreover for each $i < n$ it says whether computation at that *level* has been completed, as expressed by the different typing for c_V and c_M. The refined annotations of term-constructors (and computational types) allow to distinguish the following situations:

- $(\lambda_M x.e, E)$ we start generating code for a λ-abstraction
- $(e, E[\lambda_M x.\Box])$ we have generated a fresh name x, and start generating code for the body
- $(e, E[\lambda_M x.E'])$ we are somewhere in the middle of the computation generating code for the body
- $(ret(e), E[\lambda_M x.\Box])$ we have the code for the body of the λ-abstraction
- $(ret(\lambda_V x.e), E)$ we have the code for the λ-abstraction

All operational semantics proposed for MetaML or $\lambda^{\bigcirc}$ do not make these fine-grain distinctions. Only [10], which extends $\lambda^{\Box}$ of [5] with names a la FreshML (and *intensional analysis*), has an operational semantics with steps modeling fresh name generation and recapturing, but its relations with $\lambda^{\bigcirc}$ and MetaML have not been investigated, yet.

The *up* and *dn* primitives of MMML are related to cross-stage persistence $\%e$ and code execution run e of MetaML. In MMML demotion $vc\downarrow$ is partial, thus

evaluation of $dn(vc)$ may raise an unresolved link error, while in MetaML demotion is total, and an unresolved link error is raised only when evaluating x (at level 0). However, in [3] demotion is applied only to closed values, during evaluation of well-typed programs.

Multi-lingual extensions. It is easy to extend a monadic metalanguage, like MMML, to cope with a variety of programming languages: each programming language PL_i is modeled by a different monad M_i with its own set of operations. However, one should continue to have **one code type** constructor $\langle\tau\rangle$, i.e. , the representation of terms should be uniform. Therefore, there should be one $up\colon\tau\Rightarrow\langle\tau\rangle$ and one c_V (for each c), but several $dn_i\colon\langle\tau\rangle\Rightarrow M_i\tau$ and c_{M_i}, one for each monad M_i. In this way, we could have terms of type $M_1\langle M_2\tau\rangle$, which correspond to a program written in PL_1 for generating programs written in PL_2.

Compilation strategies. The compilation step (C.1) in Table 3 uses the demotion operation of Definition 2, which returns the term $vc\downarrow$ of type τ represented by a code value vc of type $\langle\tau\rangle$ (if such a term exists). One could adopt a lazier compilation strategy, which delays the compilation of parts of the code. A lazy strategy has the effect of delaying unresolved link errors, including the possibility of never raising them (when part of the code is dead). For instance, a possible clause for *lazy demotion* is $ret_V(e)\downarrow = dn(e)$. A more aggressive approach is to replace the compilation step with simplification rules

$$dn(up(e)) \longrightarrow e \qquad dn(c_V([\overline{x}_i]e_i|i\in m)) \longrightarrow c([\overline{x}_i]dn(e_i[\overline{x}_i := up(\overline{x}_i)])|i\in m)$$

However, one must modify the type system to ensure the SR and progress property,but changing the type schema for dn to $\langle\tau\rangle\Rightarrow\tau$ is not enough!

Type systems. We have adopted a simple type system for MMML, which does not detect statically all run-time errors. In particular, we have not included the closed type constructor $[\tau]$ of MetaML for two reasons:

1. there are alternative approaches to prevent link errors *incomparable* with the closed type approach (e.g. the *region-based* approach of [17] and the *environment classifier* approach of [11])
2. it requires *dead-code annotations* $(x)e$ that are instrumental to the proof of type safety.

Better type systems are desirable not only for detecting errors statically, but also to provide more accurate type schema for dn, e.g. $dn\colon[\langle\tau\rangle]\Rightarrow\tau$, which could justify replacing the *compilation step* by local simplification rules (see above). [10] is the best attempt up-to-date in addressing typing issues, although it does not explicitly consider computational effects. The adaptation of Nanevski's type system to MMML, e.g. refining code types $\langle\tau|C\rangle$ with a set C of names, is a subject for further research. Also the type system of [11] (where one has several code type constructors $\langle\tau\rangle^\alpha$, corresponding to different ways of representing

terms) could be adapted to MMML, but at a preliminary check it seems that the more accurate type schema $(\forall\alpha.\langle\tau\rangle^{\alpha})\Rightarrow\forall\alpha.\tau$ for *dn* is insufficient to validate the local simplification rules for compilation.

Uniform representation in Logical Frameworks. The code types of MMML provide a uniform representation of terms, similar to the (weak) Higher-Order Abstract Syntax (HOAS) encoding of object logics in a logical framework (LF). Of course, in a LF there are stronger requirements on HOAS encodings, but any advance in the area of LF is likely to advance the state-of-the-art in metaprogramming. Recently [10] has made significant advances in the area of *intensional analysis*, i.e. , the ability to analyze code (see [14]), by building on [13].

Monadic intermediate languages. [1] advocates the use of MIL for expressing optimizing transformations. Also MMML could be used for this purpose, but for having non-trivial optimizations one has to introduce more aggressive simplifications (than those strictly needed for defining the operational semantics) and refine monadic types with effect information as done in [1]. In general, we expect β-conversion $@(\lambda([x]e_2), e_1) \approx e_2[x{:=}e_1]$ and the following equivalences to be *observationally sound*

- $do(ret(e_1), [x]e_2) \approx e_2[x{:=}e_1]$
- $c_M([\overline{x}_i]ret(e_i)|i \in m) \approx ret(c_V([\overline{x}_i]e_i|i \in m))$

while other equivalences, like $@_V(\lambda_V([x]e_2), e_1) \approx e_2[x{:=}e_1]$, are more fragile (e.g. they fail when the language is extended with intensional analysis).

Acknowledgments. We thank Francois Pottier, Amr Sabry, Walid Taha for useful discussions, and the anonymous referees for their valuable comments.

References

[1] N. Benton and A. Kennedy. Monads, effects and transformations. In *Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS-99)*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, September 1999. Elsevier.

[2] G. Berry and G. Boudol. The chemical abstract machine. In *Conf. Record 17th ACM Symp. on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990*, pages 81–94. ACM Press, New York, 1990.

[3] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, to appear.

[4] 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.

[5] R. Davies and F. Pfenning. A modal analysis of staged computation. In *the Symposium on Principles of Programming Languages (POPL '96)*, pages 258–270, St. Petersburg Beach, 1996.

[6] A. Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'01, Krakow, Poland, 2–5 May 2001*, volume 2044 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, Berlin, 2001.

[7] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.

[8] J. W. Klop. *Combinatory Reduction Systems*. PhD thesis, University of Utrecht, 1980. Published as Mathematical Center Tract 129.

[9] The MetaML Home Page, 2000. Provides source code and documentation online at http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html.

[10] A. 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.

[11] M. F. Nielsen and W. Taha. Environment classifiers. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, N.Y., January 15–17 2003. ACM Press.

[12] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, and et. al. Haskell 1.4: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, Mar 1997. World Wide Web version at `http://haskell.cs.yale.edu/haskell-report`.

[13] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Programme Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, 2000.

[14] 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.

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

[16] W. Taha and T. Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.

[17] P. Thiemann and D. Dussart. Partial evaluation for higher-order languages with state, 1999. Available from `http://www.informatik.uni-freiburg.de/~thiemann/papers/index.html`

[18] P. Wadler. The marriage of effects and monads. In *the International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 63–74. ACM, June 1999.

[19] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.