An Abstract Monadic Semantics for Value Recursion

Eugenio Moggi*
DISI
Univ. di Genova
moggi@disi.unige.it

Amr Sabry†
Department of Computer Science
Indiana University
sabry@indiana.edu

Abstract

This paper proposes an operational semantics for value recursion in the context of monadic metalanguages. Our technique for combining value recursion with computational effects works *uniformly* for all monads. The operational nature of our approach is related to the implementation of recursion in Scheme and its monadic version proposed by Friedman and Sabry, but it defines a different semantics and does not rely on assignments. When contrasted to the axiomatic approach proposed by Erkök and Launchbury, our semantics for the continuation monad invalidates one of the axioms, adding to the evidence that this axiom is problematic in the presence of continuations.

1 Introduction

How should recursive definitions interact with computational effects like assignments and jumps? Consider a term *fix* x.e where *fix* is some fixed point operator and e is an expression whose evaluation has side-effects. There are at least two natural meanings for the term:

1. the term is equivalent to the unfolding $e\{x = \mathit{fix}\, x.e\}$, and the side-effects are duplicated by the unfolding.
2. the side-effects are performed the first time e is evaluated to a value v and then the term becomes equivalent to the unfolding $v\{x = \mathit{fix}\, x.v\}$.

The first meaning corresponds to the standard mathematical view [Bar84]. The second meaning corresponds to the standard operational view defined since the SECD machine [Lan64] and as implemented in Scheme for example [KCE98]. The two meanings are observationally equivalent in a pure functional language. When the computational effects are expressed using monads, Erkök and Launchbury [Erk02, EL00, ELM02] introduced the phrase *value recursion in monadic computations* for the second meaning and the name *mfix* for the corresponding fixed-point operator. Since we also work in the context of monadic metalanguages, we adopt the same terminology but use the capitalized name *Mfix* to distinguish our approach.

We propose a simple and uniform operational technique for combining monadic effects with value recursion. Computing the result of *Mfix* $x.e$ requires three rules:

1. A rule to initiate the computation of e. Since this computation happens under a binder, care must be taken to rename any other bound instance of x that we might later encounter.
2. If the computation of e returns a value v, all free occurrences of x are replaced by *fix* $x.v$ (where *fix* is the standard mathematical fixed-point operator).
3. If the computation of e attempts to use x, we signal an error.

The three rules above are robust in the sense that they can be uniformly applied to a wide range of monads: we give examples for the monads of state, non-determinism, parallelism, and continuations.

Our semantics is operational in nature but unlike the SECD and Scheme semantics, it doesn't rely on assignments to realize the second rule. The presence of assignments in the other operational approaches yields a different semantics, complicates reasoning, and invalidates some equational axioms.

*Supported by MIUR project NAPOLI and EU project DART IST-2001-33477.
†This material is based upon work supported by the NSF under Grants No. CCR 0196063 and CCR 0204389.

$$x\ \frac{\Gamma(x)=\tau}{\Gamma\vdash_\Sigma x\colon\tau}\qquad \text{abs}\ \frac{\Gamma,x\colon\tau_1\vdash_\Sigma e\colon\tau_2}{\Gamma\vdash_\Sigma \lambda x.e\colon\tau_1\to\tau_2}\qquad \text{app}\ \frac{\Gamma\vdash_\Sigma e_1\colon\tau_1\to\tau_2\quad\Gamma\vdash_\Sigma e_2\colon\tau_1}{\Gamma\vdash_\Sigma e_1e_2\colon\tau_2}$$

$$\text{ret}\ \frac{\Gamma\vdash_\Sigma e\colon\tau}{\Gamma\vdash_\Sigma \mathit{ret}\ e\colon M\tau}\qquad \text{do}\ \frac{\Gamma\vdash_\Sigma e_1\colon M\tau_1\quad\Gamma,x\colon\tau_1\vdash_\Sigma e_2\colon M\tau_2}{\Gamma\vdash_\Sigma \mathit{do}\ x\leftarrow e_1;e_2\colon M\tau_2}$$

$$l\ \frac{\Sigma(l)=R\tau}{\Gamma\vdash_\Sigma l\colon R\tau}\qquad \text{new}\ \frac{\Gamma\vdash_\Sigma e\colon\tau}{\Gamma\vdash_\Sigma \mathit{new}\ e\colon M(R\tau)}\qquad \text{get}\ \frac{\Gamma\vdash_\Sigma e\colon R\tau}{\Gamma\vdash_\Sigma \mathit{get}\ e\colon M\tau}\qquad \text{set}\ \frac{\Gamma\vdash_\Sigma e_1\colon R\tau\quad\Gamma\vdash_\Sigma e_2\colon\tau}{\Gamma\vdash_\Sigma \mathit{set}\ e_1\ e_2\colon M(R\tau)}$$

Table 1: Type System for MML^S

In contrast, the work by Erkök and Launchbury [EL00, Erk02] advocates an axiomatic approach to defining value recursion by proposing several desirable axioms. In their approach one has to find for each given monad over some category (or defined in Haskell [Jon99]) a fixed point operator that satisfy the axioms (up to observational equivalence). The endeavor has to be repeated for each monad individually. For the continuation monad there are no known fixed point operators that satisfy all the desired axioms.

Summary. Sections 2 and 3 illustrate the technique by taking an existing monadic metalanguage MML^S with ML-style references [MF03, Sec.3] and extending it with value recursion. Section 4 recalls the equational axioms for value recursion in [Erk02], and when they are known to fail. Section 5 shows that the addition of value recursion to MML^S is robust with respect to the addition of other computational effects, namely non-determinism and parallelism. Finally, Section 6 explains the full subtleties of value recursion in the presence of continuations, outlines a proof of type safety, and discusses counter-examples to equational axioms.

2 A Monadic Metalanguage with References

We introduce a monadic metalanguage MML^S for imperative computations, namely a subset of Haskell with the IO-monad. Its operational semantics is given according to the general pattern proposed in [MF03], i.e. we specify a confluent *simplification* relation $\longrightarrow$ (defined as the *compatible closure* of a set of rewrite rules), and a *computation* relation $\longmapsto$ describing how the *configurations* of the (closed) system may evolve. 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* (i.e. have no effects). 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 [WF94]), while for computationally irrelevant term-constructors it suffices to give local simplification rules, that can be applied non-deterministically (because they are semantic preserving).
The syntax of MML^S is abstracted over basic types b, variables $x\in\mathsf{X}$, and locations $l\in\mathsf{L}$.

- Types $\boxed{\tau\in\mathsf{T}::=b\mid\tau_1\to\tau_2\mid M\tau\mid R\tau}$
- Terms $\boxed{e\in\mathsf{E}\ ::=\ x\mid\lambda x.e\mid e_1e_2\mid \mathit{ret}\ e\mid \mathit{do}\ x\leftarrow e_1;e_2\mid l\mid \mathit{new}\ e\mid \mathit{get}\ e\mid \mathit{set}\ e_1\ e_2}$

In addition to the basic types, we have function types $\tau_1\to\tau_2$, reference types $R\tau$ for locations containing values of type τ, and computational types $M\tau$ for (effect-full) programs computing values of type τ. The terms $\mathit{do}\ x\leftarrow e_1;e_2$ and $\mathit{ret}\ e$ are used to sequence and terminate computations, the other monadic operations are: $\mathit{new}\ e$ which creates a new reference, $\mathit{get}\ e$ which returns the contents of a reference, and $\mathit{set}\ e_1\ e_2$ which updates the contents of reference e_1 to be e_2. In order to specify the semantics of the language, the set of terms also includes locations l.
Table 1 gives the typing rules (for deriving judgments of the form $\Gamma\vdash_\Sigma e\colon\tau$, where $\Gamma\colon\mathsf{X}\xrightarrow{fin}\mathsf{T}$ is a type assignment for variables $x\colon\tau$ and $\Sigma\colon\mathsf{L}\xrightarrow{fin}\mathsf{T}$ is a signature for locations $l\colon R\tau$.
The operational semantics is given by two relations (as outlined above): a *simplification* relation for pure evaluation and a *computation* relation for monadic evaluation. Simplification $\longrightarrow$ is given by β-reduction,

Administrative steps	
(A.0)	$(\mu, \mathit{ret}\ e, \Box) \longmapsto \mathsf{done}$
(A.1)	$(\mu, \mathit{do}\ x \leftarrow e_1; e_2, E) \longmapsto (\mu, e_1, E[\mathit{do}\ x \leftarrow \Box; e_2])$
(A.2)	$(\mu, \mathit{ret}\ e_1, E[\mathit{do}\ x \leftarrow \Box; e_2]) \longmapsto (\mu, e_2\{x{:=}e_1\}, E)$
Imperative steps	
(new)	$(\mu, \mathit{new}\ e, E) \longmapsto (\mu\{l{:}\,e\}, \mathit{ret}\ l, E)$ where $l \notin \mathsf{dom}(\mu)$
(get)	$(\mu, \mathit{get}\ l, E) \longmapsto (\mu, \mathit{ret}\ e, E)$ with $e = \mu(l)$
(set)	$(\mu, \mathit{set}\ l\ e, E) \longmapsto (\mu\{l = e\}, \mathit{ret}\ l, E)$ with $l \in \mathsf{dom}(\mu)$

Table 2: Computation Relation for MML^S

i.e. the compatible closure of $(\lambda x.e_2)e_1 \longrightarrow e_2\{x{:=}e_1\}$. The computation relation $Id \longmapsto Id' \mid \mathsf{done}$ (see Table 2) is defined using the additional notions of evaluation contexts, stores and configurations $Id \in \mathsf{Conf}$:

- Evaluation contexts $\boxed{E \in \mathsf{EC} ::= \Box \mid E[\mathit{do}\ x \leftarrow \Box; e]}$ (or equivalently $E ::= \Box \mid \mathit{do}\ x \leftarrow E; e$).
- Stores $\mu \in \mathsf{S} \triangleq \mathsf{L} \xrightarrow{\mathit{fin}} \mathsf{E}$ map locations to their contents.
- Configurations $(\mu, e, E) \in \mathsf{Conf} \triangleq \mathsf{S} \times \mathsf{E} \times \mathsf{EC}$ consist of the current store μ, the program fragment e under consideration, and its evaluation context E.

3 Extension with Value Recursion

We now describe the monadic metalanguage $\mathsf{MML}^S_{\mathit{fix}}$ obtained by extending MML^S with two fixed point constructs: *fix* $x.e$ for ordinary recursion, and *Mfix* $x.e$ for value recursion. The expression *fix* $x.e$ *simplifies* to its unfolding. For *computing* the value of *Mfix* $x.e$, the subexpression e is first evaluated to a monadic value *ret* e'. This evaluation might perform computational effects but cannot use x. Then all occurrences of x in e' are bound to the monadic value itself using *fix* so that any unfolding will not redo the computational effects. The extension $\mathsf{MML}^S_{\mathit{fix}}$ is an *instance* of a general pattern (only the extension of the computation relation is non-trivial), that will become clearer after considering other monadic metalanguages.

- Terms $\boxed{e \in \mathsf{E} \mathrel{+}= \mathit{fix}\ x.e \mid \mathit{Mfix}\ x.e}$
- Evaluation contexts $\boxed{E \in \mathsf{EC} \mathrel{+}= E[\mathit{Mfix}\ x.\Box]}$
- Configurations $(X|\mu, e, E) \in \mathsf{Conf} \triangleq \mathcal{P}_{\mathit{fin}}(\mathsf{X}) \times \mathsf{S} \times \mathsf{E} \times \mathsf{EC}$. The additional component X is a set which records the recursive variables generated so far, thus X grows as the computation progresses.

Despite their different semantics, the two fixed points have similar typing rules:

$$\frac{\Gamma, x{:}\,M\tau \vdash_\Sigma e{:}\,M\tau}{\Gamma \vdash_\Sigma \mathit{fix}\ x.e{:}\,M\tau} \qquad\qquad \frac{\Gamma, x{:}\,M\tau \vdash_\Sigma e{:}\,M\tau}{\Gamma \vdash_\Sigma \mathit{Mfix}\ x.e{:}\,M\tau}$$

The simplification relation is extended with the *fix*-unfolding rule $\mathit{fix}\ x.e \longrightarrow e\{x{:=}\mathit{fix}\ x.e\}$.
The computation relation $Id \longmapsto Id' \mid \mathsf{done} \mid \mathsf{err}$ may now raise an error and is defined by the following rules:

- the rules in Table 2, modified to propagate the set X unchanged, and
- the following new rules for evaluating recursive monadic bindings *Mfix* $x.e$:

(M.1) $(X|\mu, \mathit{Mfix}\ x.e, E) \longmapsto (X, x|\mu, e, E[\mathit{Mfix}\ x.\Box])$ with x renamed to avoid clashes with X

(M.2) $(X|\mu, \mathit{ret}\ e, E[\mathit{Mfix}\ x.\Box]) \longmapsto (X|\overline{\mu}, \mathit{ret}\ \overline{e}, \overline{E})$ where $\overline{\bullet}$ stands for $\bullet\{x := \mathit{fix}\ x.\mathit{ret}\ e\}$

(err) $(X|\mu, x, E) \longmapsto \mathsf{err}$ where $x \in X$ (attempt to use an unresolved recursive variable)

In the context $\mathit{Mfix}\ x.\Box$ the hole is within the scope of a binder, thus it requires evaluation of open terms:

- The rule (M.1) behaves like *gensym*, it ensures *freshness* of x. As the computation progresses x may leak anywhere in the configuration (depending on the computational effects available in the language).
- The rule (M.2) does the reverse, it replaces all *free occurrences* of x in the configuration with the term $\mathit{fix}\ x.\mathit{ret}\ e$, in which x is not free. This rule is quite subtle, because of $E\{x := e\}$ (see Definition 6.5).

In special cases [AFMZ02] it is possible to simplify (M.2) by treating X as a stack and enforcing the invariant that $\mathrm{FV}(E) = \emptyset$, but our aim is an operational semantics that works with *arbitrary* computational effects. Indeed in the case of continuations (Section 6), neither of these invariants holds.

4 Axioms for Value Recursion

In [Erk02] the fixed point constructs have a slightly different typing:

- $\dfrac{\Gamma, x{:}\,\tau \vdash_\Sigma e{:}\, M\tau}{\Gamma \vdash_\Sigma \mathit{mfix}\ x.e{:}\, M\tau}$ where x is of type τ.

  This rule allows the use of x at type τ before the recursion is resolved, as in $(\mathit{mfix}\ x.\mathit{set}\ x\ 0){:}\, M(R\ \mathit{int})$. In [Erk02] this premature attempt to use x is identified with *divergence*, while but we consider it a *monadic error*, which should be statically prevented by more refined type systems [Bou01]. The difference of typing reflects this desire and is not an intrinsic limitation of our approach.

- $\dfrac{\Gamma, x{:}\,\tau \vdash_\Sigma e{:}\,\tau}{\Gamma \vdash_\Sigma \mathit{fix}\ x.e{:}\,\tau}$ requires recursive definitions at *all types*; we only require them at *computational types*.

Two of the most notable proposed axioms for defining value recursion in [Erk02] are:

(Purity)	$\mathit{mfix}\ x.\mathit{ret}\ e$	$=$	$\mathit{ret}\ (\mathit{fix}\ x.e)$	
(Left-shrinking)	$\mathit{mfix}\ x.(\mathit{do}\ x_1 \leftarrow e_1; e_2)$	$=$	$\mathit{do}\ x_1 \leftarrow e_1; \mathit{mfix}\ x.e_2$	when $x \notin \mathrm{FV}(e_1)$

The *purity* axiom requires that *mfix* coincides with *fix* for pure computations. Because of the differences in typing, the *purity* axiom in our case becomes:

$$\text{(Purity)} \qquad \mathit{Mfix}\ x.\mathit{ret}\ e \quad = \quad \mathit{fix}\ x.\mathit{ret}\ e$$

Left-shrinking states that computations which do not refer to the recursive variable can be moved outside the recursive definition. This rewriting however is known to be incorrect in Scheme [Baw88] but it was argued [Erk02] that the failure of left-shrinking is due to the idiosyncrasies of Scheme. In fact left-shrinking is invalidated by our semantics and in other known combinations of value recursion and continuations [FS00, Car03]. Indeed if one captures the continuation in e_1 then on the left-hand side this continuation has access to free occurrences of x in e_2 but not on the right-hand side. As Section 6.2 illustrates this can be exploited to write a counterexample to left-shrinking.

5 Extensions with Non-Determinism and Parallelism

We consider two extensions to MML^S (and $\mathsf{MML}^S_{\mathit{fix}}$): the first introduces non-deterministic choice e_1 *or* e_2, the second introduces a construct *spawn* $e_1\ e_2$ to spawn a thread of computation e_1 in parallel with the continuation e_2 of the current thread.

Non-determinism. The typing rule for non-deterministic choice is:

$$\frac{\Gamma \vdash_\Sigma e_1 : M\tau \quad \Gamma \vdash_\Sigma e_2 : M\tau}{\Gamma \vdash_\Sigma e_1 \ or \ e_2 : M\tau}$$

The configurations for MML^S and MML^S_{fix} are unchanged. The computation relations are modified to become non-deterministic. More specifically,

- for MML^S, we add the computation rules $(\mu, e_1 \ or \ e_2, E) \longmapsto (\mu, e_i, E)$ for $i = 1, 2$;
- for MML^S_{fix}, we add the computation rules $(X|\mu, e_1 \ or \ e_2, E) \longmapsto (X|\mu, e_i, E)$ for $i = 1, 2$.

Parallelism. The typing rule for *spawn* is:

$$\frac{\Gamma \vdash_\Sigma e_1 : M\tau_1 \quad \Gamma \vdash_\Sigma e_2 : M\tau_2}{\Gamma \vdash_\Sigma spawn \ e_1 \ e_2 : M\tau_2}$$

In this case a configuration consists of a (finite) multi-set of parallel threads sharing the store μ, where each thread is represented by a pair (e, E).

For MML^S the configurations become $\langle \mu, N\rangle \in \mathsf{Conf} \triangleq \mathsf{S} \times \mathcal{M}_{fin}(\mathsf{E} \times \mathsf{EC})$, i.e. instead of a thread (e, E) one has a multi-set of threads, and the computation relation $Id \longmapsto Id' \mid \mathsf{done}$ is defined by the following rules:

- Administrative steps: threads act independently, termination occurs when all threads have completed

(done) $\langle \mu, \emptyset\rangle \longmapsto \mathsf{done}$

(A.0) $\langle \mu, (ret \ e, \Box) \uplus N\rangle \longmapsto \langle \mu, N\rangle$

(A.1) $\langle \mu, (do \ x \leftarrow e_1; e_2, E) \uplus N\rangle \longmapsto \langle \mu, (e_1, E[do \ x \leftarrow \Box; e_2]) \uplus N\rangle$

(A.2) $\langle \mu, (ret \ e_1, E[do \ x \leftarrow \Box; e_2]) \uplus N\rangle \longmapsto \langle \mu, (e_2\{x := e_1\}, E) \uplus N\rangle$

- Imperative steps: each thread can operate on the shared store

(new) $\langle \mu, (new \ e, E) \uplus N\rangle \longmapsto \langle \mu\{l : e\}, (ret \ l, E) \uplus N\rangle$ where $l \notin \mathsf{dom}(\mu)$

(get) $\langle \mu, (get \ l, E) \uplus N\rangle \longmapsto \langle \mu, (ret \ e, E) \uplus N\rangle$ with $e = \mu(l)$

(set) $\langle \mu, (set \ l \ e, E) \uplus N\rangle \longmapsto \langle \mu\{l = e\}, (ret \ l, E) \uplus N\rangle$ with $l \in \mathsf{dom}(\mu)$

- Step for spawning a new thread

(spawn) $\langle \mu, (spawn \ e_1 \ e_2, E) \uplus N\rangle \longmapsto \langle \mu, (e_1, \Box) \uplus (e_2, E) \uplus N\rangle$

For MML^S_{fix} the configurations become $\langle X|\mu, N\rangle \in \mathsf{Conf} \triangleq \mathcal{P}_{fin}(\mathsf{X}) \times \mathsf{S} \times \mu(\mathsf{E} \times \mathsf{EC})$, i.e. the threads share the set X which records the recursive variables generated so far, and the computation relation $Id \longmapsto Id' \mid \mathsf{done} \mid \mathsf{err}$ is defined by the rules above (modified to propagate the set X unchanged) and the following rules for recursive monadic bindings:

(M.1) $\langle X|\mu, (Mfix \ x.e, E) \uplus N\rangle \longmapsto \langle X, x|\mu, (e, E[Mfix \ x.\Box]) \uplus N\rangle$ with x renamed to avoid clashes with X

(M.2) $\langle X|\mu, (ret \ e, E[Mfix \ x.\Box]) \uplus N\rangle \longmapsto \langle X|\overline{\mu}, (ret \ \overline{e}, \overline{E}) \uplus \overline{N}\rangle$ where $\overline{\bullet}$ stands for $\bullet\{x := fix \ x.ret \ e\}$

(err) $\langle X|\mu, (x, E) \uplus N\rangle \longmapsto \mathsf{err}$ where $x \in X$

When a thread resolves a recursive variable x (M.2), the value of x is propagated to all other threads. When an error occurs in a thread (err), the whole computation crashes.

$$x\ \frac{\Gamma(x)=\tau}{\Gamma\vdash_\Sigma x\colon\tau}\qquad \text{abs}\ \frac{\Gamma,x\colon\tau_1\vdash_\Sigma e\colon\tau_2}{\Gamma\vdash_\Sigma \lambda x.e\colon\tau_1\to\tau_2}\qquad \text{app}\ \frac{\Gamma\vdash_\Sigma e_1\colon\tau_1\to\tau_2\quad\Gamma\vdash_\Sigma e_2\colon\tau_1}{\Gamma\vdash_\Sigma e_1e_2\colon\tau_2}\qquad \text{fix}\ \frac{\Gamma,x\colon M\tau\vdash_\Sigma e\colon M\tau}{\Gamma\vdash_\Sigma \mathit{fix}\ x.e\colon M\tau}$$

$$\text{ret}\ \frac{\Gamma\vdash_\Sigma e\colon\tau}{\Gamma\vdash_\Sigma \mathit{ret}\ e\colon M\tau}\qquad \text{do}\ \frac{\Gamma\vdash_\Sigma e_1\colon M\tau_1\quad\Gamma,x\colon\tau_1\vdash_\Sigma e_2\colon M\tau_2}{\Gamma\vdash_\Sigma \mathit{do}\ x\leftarrow e_1;e_2\colon M\tau_2}\qquad \text{Mfix}\ \frac{\Gamma,x\colon M\tau\vdash_\Sigma e\colon M\tau}{\Gamma\vdash_\Sigma \mathit{Mfix}\ x.e\colon M\tau}$$

$$l\ \frac{\Sigma(l)=R\tau}{\Gamma\vdash_\Sigma l\colon R\tau}\qquad \text{new}\ \frac{\Gamma\vdash_\Sigma e\colon\tau}{\Gamma\vdash_\Sigma \mathit{new}\ e\colon M(R\tau)}\qquad \text{get}\ \frac{\Gamma\vdash_\Sigma e\colon R\tau}{\Gamma\vdash_\Sigma \mathit{get}\ e\colon M\tau}\qquad \text{set}\ \frac{\Gamma\vdash_\Sigma e_1\colon R\tau\quad\Gamma\vdash_\Sigma e_2\colon\tau}{\Gamma\vdash_\Sigma \mathit{set}\ e_1\ e_2\colon M(R\tau)}$$

$$k\ \frac{\Sigma(k)=K\tau}{\Gamma\vdash_\Sigma k\colon K\tau}\qquad \text{callcc}\ \frac{\Gamma,x\colon K\tau\vdash_\Sigma e\colon M\tau}{\Gamma\vdash_\Sigma \mathit{callcc}\ x.e\colon M\tau}\qquad \text{throw}\ \frac{\Gamma\vdash_\Sigma e_1\colon K\tau\quad\Gamma\vdash_\Sigma e_2\colon M\tau}{\Gamma\vdash_\Sigma \mathit{throw}\ e_1\ e_2\colon M\tau'}$$

Table 3: Type System for MML^{SK}_{fix}

6 A Monadic Metalanguage with References and Continuations

In this section we consider in full detail the monadic metalanguage MML^{SK}_{fix}, obtained from MML^{S}_{fix} by adding continuations. Section 6.1 outlines a proof of type safety, and Section 6.2 shows the failure of the left-shrinking axiom and discusses some differences with Scheme. The syntax of MML^{SK}_{fix} is abstracted over basic types b, variables $x\in\mathsf{X}$, locations $l\in\mathsf{L}$ and continuations $k\in\mathsf{K}$:

- Types $\boxed{\tau\in\mathsf{T}::=b\mid\tau_1\to\tau_2\mid M\tau\mid R\tau\mid K\tau}$
- Terms $\boxed{\begin{array}{rcl} e\in\mathsf{E} & ::= & x\mid\lambda x.e\mid e_1e_2\mid \mathit{fix}\ x.e\mid \mathit{ret}\ e\mid \mathit{do}\ x\leftarrow e_1;e_2\mid \mathit{Mfix}\ x.e\mid\\ & & l\mid \mathit{new}\ e\mid \mathit{get}\ e\mid \mathit{set}\ e_1\ e_2\mid k\mid \mathit{callcc}\ x.e\mid \mathit{throw}\ e_1e_2\end{array}}$

The type $K\tau$ is the type of continuations which can be invoked on arguments of type $M\tau$ (invoking the continuation aborts the current context). The expression *callcc* $x.e$ binds the current continuation to x; the expression *throw* e_1e_2 has the dual effect of aborting the current continuation and using e_1 instead as the current continuation. This effectively "jumps" to the point where the continuation e_1 was captured by *callcc*.

Table 3 gives the typing rules for deriving judgments of the form $\Gamma\vdash_\Sigma e\colon\tau$, where $\Gamma\colon\mathsf{X}\xrightarrow{fin}\mathsf{T}$ is a type assignment for variables $x\colon\tau$ and $\Sigma\colon\mathsf{L}\cup\mathsf{K}\xrightarrow{fin}\mathsf{T}$ is a signature for locations $l\colon R\tau$ and continuations $k\colon K\tau$.

The *simplification* relation $\longrightarrow$ on terms is given by the compatible closure of the following rewrite rules:

β) $(\lambda x.e_2)e_1\longrightarrow e_2\{x:=e_1\}$

fix) $\mathit{fix}\ x.e\longrightarrow e\{x:=\mathit{fix}\ x.e\}$

We write $=$ for the equivalence induced by $\longrightarrow$, i.e. the reflexive, symmetric and transitive closure of $\longrightarrow$. We state the properties of simplification relevant for our purposes.

Proposition 6.1 (Congr) *The equivalence $=$ induced by $\longrightarrow$ is a congruence.*

Proposition 6.2 (CR) *The simplification relation $\longrightarrow$ is confluent.*

Proposition 6.3 (SR) *If $\Gamma\vdash_\Sigma e\colon\tau$ and $e\longrightarrow e'$, then $\Gamma\vdash_\Sigma e'\colon\tau$.*

To define the computation relation $Id\longmapsto Id'\mid\mathsf{done}\mid\mathsf{err}$ (see Table 4), we need the auxiliary notions of evaluation contexts, stores, continuation environments, configurations $Id\in\mathsf{Conf}$, and computational redexes:

- Evaluation contexts $\boxed{E\in\mathsf{EC}::=\Box\mid E[\mathit{do}\ x\leftarrow\Box;e]\mid E[\mathit{Mfix}\ x.\Box]}$ (or $E::=\Box\mid \mathit{do}\ x\leftarrow E;e\mid \mathit{Mfix}\ x.E$)
- Stores $\mu\in\mathsf{S}\triangleq\mathsf{L}\xrightarrow{fin}\mathsf{E}$ and continuation environments $\rho\in\mathsf{KE}\triangleq\mathsf{K}\xrightarrow{fin}\mathsf{EC}$
- Configurations $(X|\mu,\rho,e,E)\in\mathsf{Conf}\triangleq\mathcal{P}_{fin}(\mathsf{X})\times\mathsf{S}\times\mathsf{KE}\times\mathsf{E}\times\mathsf{EC}$ consist of the current store μ and continuation environment ρ, the program fragment e under consideration and its evaluation context E. The set X records the recursive variables generated so far, thus X grows as the computation progresses.

Administrative steps

(A.0) $(X|\mu, \rho, \mathit{ret}\ e, \Box) \longmapsto \mathsf{done}$

(A.1) $(X|\mu, \rho, \mathit{do}\ x \leftarrow e_1; e_2, E) \longmapsto (X|\mu, \rho, e_1, E[\mathit{do}\ x \leftarrow \Box; e_2])$

(A.2) $(X|\mu, \rho, \mathit{ret}\ e_1, E[\mathit{do}\ x \leftarrow \Box; e_2]) \longmapsto (X|\mu, \rho, e_2\{x{:=}\, e_1\}, E)$

Steps for recursive monadic binding

(M.1) $(X|\mu, \rho, \mathit{Mfix}\ x.e, E) \longmapsto (X, x|\mu, \rho, e, E[\mathit{Mfix}\ x.\Box])$ with x renamed to avoid clashes with X

(M.2) $(X|\mu, \rho, \mathit{ret}\ e, E[\mathit{Mfix}\ x.\Box]) \longmapsto (X|\overline{\mu}, \overline{\rho}, \mathit{ret}\ \overline{e}, \overline{E})$ where $\overline{\bullet}$ stands for $\bullet\{x{:=}\, \mathit{fix}\ x.\mathit{ret}\ e\}$
(the free occurrences of the recursive variable x are replaced anywhere in the configuration)

(err) $(X|\mu, \rho, x, E) \longmapsto \mathsf{err}$ where $x \in X$ (attempt to use an unresolved recursive variable)

Imperative steps

(new) $(X|\mu, \rho, \mathit{new}\ e, E) \longmapsto (X|\mu\{l{:}\, e\}, \rho, \mathit{ret}\ l, E)$ where $l \notin \mathsf{dom}(\mu)$

(get) $(X|\mu, \rho, \mathit{get}\ l, E) \longmapsto (X|\mu, \rho, \mathit{ret}\ e, E)$ with $e = \mu(l)$

(set) $(X|\mu, \rho, \mathit{set}\ l\ e, E) \longmapsto (X|\mu\{l = e\}, \rho, \mathit{ret}\ l, E)$ with $l \in \mathsf{dom}(\mu)$

Control steps

(callcc) $(X|\mu, \rho, \mathit{callcc}\ x.e, E) \longmapsto (X|\mu, \rho\{k{:}\, E\}, e\{x{:=}\, k\}, E)$ where $k \notin \mathsf{dom}(\rho)$

(throw) $(X|\mu, \rho, \mathit{throw}\ k\ e, E) \longmapsto (X|\mu, \rho, e, E_k)$ with $E_k = \rho(k)$

Table 4: Computation Relation for MML_{fix}^{SK}

- Computational redexes
  $r \in \mathsf{R} ::= \mathit{ret}\ e \mid \mathit{do}\ x \leftarrow e_1; e_2 \mid \mathit{Mfix}\ x.e \mid \mathit{new}\ e \mid \mathit{get}\ l \mid \mathit{set}\ l\ e \mid \mathit{callcc}\ x.e \mid \mathit{throw}\ k\ e$

Remark 6.4 In the absence of *Mfix* $x.e$, the hole $\Box$ of an evaluation context E is never within the scope of a binder. Therefore one can represent E as a λ-abstraction $\lambda x.E[x]$, where $x \notin \mathrm{FV}(E)$. This is how continuations are modeled in the λ-calculus, in particular the operation $E[e]$ of replacing the hole in E with a term e becomes simplification of the β-redex $(\lambda x.E[x])\ e$. This representation of continuations is adopted also in the reduction semantics of functional languages with control operators [WF94]. In such reduction semantics there is no need keep a continuation environment ρ, because a continuation k with $\rho(k) = E$ is represented by the λ-abstraction $\lambda x.E[x]$. In the presence of *Mfix* $x.e$ (or when modeling partial evaluation, multi-stage programming, and call-by-need [AF97, AMO+95, MOW98]), evaluation may take place within the scope of a binder, and one can no longer represent an evaluation context with a λ-abstraction, because the operation $E[e]$ may capture free variables in e. In this case, continuation environments are very convenient, since the subtle issues regarding variable capture are confined to the level of configurations, and do not percolate in terms and other syntactic categories. ∎

In an evaluation context 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 set of captured variables. Moreover, the definition of $E\{x' := e'\}$ differs from the capture-avoiding substitution $e\{x' := e'\}$ for terms, because captured variables cannot be renamed.

Definition 6.5 *The sets* $\mathrm{CV}(E)$ *and* $\mathrm{FV}(E)$ *of captured and free variables and the substitution* $E\{x' := e'\}$ *are defined by induction on* E*:*

- $\mathrm{CV}(\Box) \triangleq \mathrm{FV}(\Box) \triangleq \emptyset$ *and* $\Box\{x' := e'\} \triangleq \Box$
- $\mathrm{CV}(\mathit{do}\ x \leftarrow E; e) \triangleq \mathrm{CV}(E)$, $\mathrm{FV}(\mathit{do}\ x \leftarrow E; e) \triangleq \mathrm{FV}(E) \cup (\mathrm{FV}(e) \setminus \{x\})$ *and*
  $(\mathit{do}\ x \leftarrow E; e)\{x' := e'\} \triangleq \mathit{do}\ x \leftarrow E\{x' := e'\}; e\{x' := e'\}$
  (the bound variable x *can be renamed to be different from* x' *and from any of the free variables of* e'*).*

$$(\Box)\ \frac{}{\Delta, \Box\colon M\tau \vdash_\Sigma \Box\colon M\tau} \qquad (\mathrm{do})\ \frac{\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau_1 \quad \Delta, x\colon \tau_1 \vdash_\Sigma e\colon M\tau_2}{\Delta, \Box\colon M\tau \vdash_\Sigma do\ x \leftarrow E; e\colon M\tau_2}$$

$$(\mathrm{Mfix})\ \frac{\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'}{\Delta, \Box\colon M\tau \vdash_\Sigma Mfix\ x.E\colon M\tau'}\ \Delta(x) = M\tau'$$

Table 5: Well-formed Evaluation Contexts for MML^{SK}_{fix}

- $\mathrm{CV}(Mfix\ x.E) \triangleq \{x\} \cup \mathrm{CV}(E)$, $\mathrm{FV}(Mfix\ x.E) \triangleq \mathrm{FV}(E) \setminus \{x\}$ *and*
  $(Mfix\ x.E)\{x' := e'\} \triangleq \begin{cases} Mfix\ x.E & \text{if } x = x' \\ Mfix\ x.E\{x' := e'\} & \text{otherwise} \end{cases}$
  (the captured variable x cannot be renamed; free occurrences of x in e' may be captured.)

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

Lemma 6.6

1. *If* $(X|\mu, \rho, e, E) \longrightarrow (X'|\mu', \rho', e', E')$, *then* $X = X'$, $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$, $\mathsf{dom}(\rho') = \mathsf{dom}(\rho)$ *and*
   - $\mathrm{FV}(e') \subseteq \mathrm{FV}(e)$, $\mathrm{CV}(E') = \mathrm{CV}(E)$ *and* $\mathrm{FV}(E') \subseteq \mathrm{FV}(E)$
   - $\mathrm{FV}(\mu'\ l) \subseteq \mathrm{FV}(\mu\ l)$ *for* $l \in \mathsf{dom}(\mu)$
   - $\mathrm{CV}(\rho'\ k) = \mathrm{CV}(\rho\ k)$ *and* $\mathrm{FV}(\rho'\ k) \subseteq \mathrm{FV}(\rho\ k)$ *for* $k \in \mathsf{dom}(\rho)$
2. *If* $(X|\mu, \rho, e, E) \longmapsto (X'|\mu', \rho', e', E')$ *and* $\mathrm{FV}(\mu, \rho, e, E) \cup \mathrm{CV}(\rho, E) \subseteq X$, *then*
   $X \subseteq X'$, $\mathsf{dom}(\mu) \subseteq \mathsf{dom}(\mu')$, $\mathsf{dom}(\rho) \subseteq \mathsf{dom}(\rho')$ *and* $\mathrm{FV}(\mu', \rho', e', E') \cup \mathrm{CV}(\rho', E') \subseteq X'$.

Theorem 6.7 (Bisim) *If* $Id \equiv (X|\mu, \rho, e, E)$ *with* $e \in \mathsf{R}$ *and* $Id \xrightarrow{*} Id'$, *then*

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

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

6.1 Type Safety

The definitions of well-formed configurations $\Delta \vdash_\Sigma Id\colon \tau'$ and evaluation contexts $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ must take into account the set X. Thus we need a type assignment Δ mapping $x \in X$ to computational types $M\tau$.

Definition 6.8 $\Delta \vdash_\Sigma (X|\mu, \rho, e, E)\colon \tau' \overset{\Delta}{\Longleftrightarrow} \mathsf{dom}(\Sigma) = \mathsf{dom}(\mu) \uplus \mathsf{dom}(\rho)$, $\mathsf{dom}(\Delta) = X$ *and exists* τ *such that*

- $\Delta \vdash_\Sigma e\colon M\tau$ *is derivable*
- $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ *is derivable (see Table 5)*
- $e_l = \mu(l)$ *and* $R\tau_l = \Sigma(l)$ *implies* $\Delta \vdash_\Sigma e_l\colon \tau_l$
- $E_k = \rho(k)$ *and* $K\tau_k = \Sigma(k)$ *implies* $\Delta, \Box\colon M\tau_k \vdash_\Sigma E_k\colon M\tau'$.

The formation rules of Table 5 for deriving $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ ensure that Δ assigns a computational type to all captured variables of E. We can now formulate the SR and progress properties for MML^{SK}_{fix}.

Theorem 6.9 (SR)

1. *If* $\Delta \vdash_\Sigma Id_1\colon \tau'$ *and* $Id_1 \longrightarrow Id_2$, *then* $\Delta \vdash_\Sigma Id_2\colon \tau'$
2. *If* $\Delta_1 \vdash_{\Sigma_1} Id_1\colon \tau'$ *and* $Id_1 \longmapsto Id_2$, *then exists* $\Sigma_2 \supseteq \Sigma_1$ *and* $\Delta_2 \supseteq \Delta_1$ *s.t.* $\Delta_2 \vdash_{\Sigma_2} Id_2\colon \tau'$.

Theorem 6.10 (Progress) *If* $\Delta \vdash_\Sigma (X|\mu, \rho, e, E)\colon \tau'$, *then one of the following holds*

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

6.2 Counter-examples

The left-shrinking property states that:

$$\mathit{Mfix}\ x.(\mathit{do}\ x_1 \leftarrow e_1; e_2) = \mathit{do}\ x_1 \leftarrow e_1; \mathit{Mfix}\ x.e_2 \qquad \text{when } x \notin \mathrm{FV}(e_1)$$

It is instructive to consider how this property fails in MML_{fix}^{SK}. Our example (inspired by examples by Bawden and Carlsson) uses continuations in a way that requires recursive types which can be declared as follows in Haskell syntax:

```
data XT m = Xfold (m (Int, XT m))  -- final result
data KT m = Kfold (K (RT m))       -- recursive continuations
data RT m =                        -- arguments to continuations
   Final (XT m)
 | Intermediate (Bool, KT m)
```

Now we consider the following instance of the left-hand side (again in Haskell syntax):

```
t1 =
 Mfix (\x ->
   do p <- callcc (\k -> return (Intermediate (True, Kfold k)))
      case p of
        Intermediate (b, Kfold k) ->
          if b
            then
              do Final v <- callcc (\c -> throw k (return (Intermediate (False, Kfold c))))
                 return (1,v)
            else throw k (return (Final (Xfold x))))
```

In our semantics (extended with simplification rules for booleans, pairs, etc) the example evaluates as follows. The pair `p` initially refers to a continuation which re-binds `p`. In the `then`-branch which is initially taken, this continuation is invoked with a new pair containing the continuation `c`. This latter continuation expects a value `v` which it includes in the final result `(1,v)`. In the `else`-branch which is taken the second time, that value `v` is bound to `Final (Xfold x)`. Hence the return value of the body of the *Mfix* is `(1,Final (Xfold x))` and the entire expression evaluates to the recursive pair of ones `fix x. return (1, Final (Xfold x))`. However were we to move the first *callcc*-expression (which has no free occurrences of `x`) outside the *Mfix*, the continuations `k` and `c` would have no access to the variable `x` and the example would evaluate to `return (1,x)` which would cause an error if the second component is needed. The fact that this result is an approximation of the left-hand side does not generalize: with a slightly more complicated example, it is possible to get a different observable value.

Our semantics also differs from the Scheme semantics. The difference in this case is due to the nature of variables in both systems: in our setting variable are bound to expressions and locations must be created and dereferenced explicitly. In Scheme variables implicitly refer to locations, which means that continuations captured within the body of an *Mfix* not only have access to the free occurrences of the recursive variable in the body of the recursive definition but also to the *location* in which the result is to be stored: this additional expressiveness for continuations invalidates even more transformations like $\mathit{Mfix}\ x.e = e$ when $x \notin FV(e)$ [Baw88]. Such transformations should still be valid in our model.

Acknowledgments

We would like to thank Levent Erkök and Magnus Carlsson for very fruitful discussions and comments.

References

[AF97] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, May 1997.

[AFMZ02] D. Ancona, S. Fagorzi, E. Moggi, and E. Zucca. Mixin modules and computational effects. Submitted, 2002.

[AMO+95] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, pages 233–246, New York, NY, USA, 1995. ACM Press.

[Bar84] H[endrik] P[ieter] Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[Baw88] Alan Bawden. Letrec and callcc implement references. Message to `comp.lang.scheme`, 1988.

[Bou01] Gérard Boudol. The recursive record semantics of objects revisited. *Lecture Notes in Computer Science*, 2028:269–283, 2001.

[Car03] Magnus Carlsson. Value recursion in the continuation monad. Unpublished Note, January 2003.

[EL00] Levent Erkök and John Launchbury. Recursive monadic bindings. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 174–185, N.Y., September 18–21 2000. ACM Press.

[ELM02] Levent Erkök, John Launchbury, and Andrew Moran. Semantics of value recursion for monadic input/output. *Journal of Theoretical Informatics and Applications*, 36(2):155–180, 2002.

[Erk02] Levent Erkök. *Value Recursion in Monadic Computations*. PhD thesis, OGI School of Science and Engineering, OHSU, Portland, Oregon, 2002.

[FS00] Daniel P. Friedman and Amr Sabry. Recursion is a computational effect. Technical Report 546, Computer Science Department, Indiana University, December 2000.

[Jon99] Report on the programming language Haskell 98, February 1999.

[KCE98] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[MF03] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. In *FoSSaCS 2003*, LNCS. Springer-Verlag, 2003.

[MOW98] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, May 1998.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.