Theoretical Informatics and Applications Will be set by the publisher
Informatique Théorique et Applications

AN ABSTRACT MONADIC SEMANTICS FOR VALUE RECURSION *, **

EUGENIO MOGGI[1] AND AMR SABRY[2]

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.

1991 Mathematics Subject Classification. 68N18, 68Q55 .

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 [6]. The second meaning corresponds to the standard operational view defined since the SECD machine [11, 31] and as implemented in Scheme for example [30]. The two meanings

* *Supported by EU project DART IST-2001-33477 and APPSEM-II IST-2001-38957.*
** *Supported by the NSF under Grants No. CCR 0196063 and CCR 0204389.*

[1] DISI, Univ. di Genova, email: moggi@disi.unige.it
[2] Dept. of Computer Science, Indiana Univ., email: sabry@indiana.edu

are observationally equivalent in a pure functional language but not in the presence of effects. When the computational effects are expressed using monads, Erkök and Launchbury [18–20] 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.

Value recursion has applications in modeling stream-oriented computations like hardware circuits [32] and in modeling objects [9] and module systems [17,25]. We give a simple example here and refer to Section 6 for more details. In the following code fragment, f is a recursive procedure representing an object whose local state is represented by the location r:

$$\mathit{Mfix}\ f.\mathit{do}\ r \leftarrow \mathit{new}\ 0; \mathit{ret}\ (\lambda x.\ldots f \ldots r \ldots)$$

Clearly the creation of a new location r is *not* intended to be unfolded and repeated every time f is called. Instead the creation and initialization of the location r should happen the first time the body of *Mfix* is evaluated; when this evaluation produces the value $\mathit{ret}\ (\lambda x.\ldots f \ldots r \ldots)$, the recursive variable f is bound to it; subsequent calls and recursive calls to f do not re-create the location.

The example outlines a simple uniform operational technique for combining monadic effects with value recursion. Computing the result of $\mathit{Mfix}\ x.e$ 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 (in a monadic metalanguage a value is simply a term of the form $\mathit{ret}\ e'$) then all free occurrences of x are replaced by $\mathit{fix}\ x.v$ (where *fix* is the standard mathematical fixed point operator). Naturally the computation of e may perform computational effects but it cannot use x.
(3) If the computation of e attempts to use x, we signal an error.

In the second rule, the notion that "e returns a value v" is quite informal at this point. Indeed depending on the computational effects in question, e might terminate without returning a proper value (in the case of exceptions), or it might return more than one value (in the case of non-determinism), or it might return more than once (in the case of continuations). One of the main benefits of our abstract approach is that it provides a formal semantics for value recursion in all these situations. Indeed we show that the rules above are robust in the sense that they can be uniformly applied to a wide range of monads giving a semantics for value recursion in every case: we give examples for the monads of state, non-determinism, exceptions, 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. Section 5 includes a discussion of this point. We point however that the question of which transformations are invalidated depends on the details of the implementation: for example, a more careful implementation of the Scheme semantics does not invalidate as many transformations as the straightforward implementation [38].

In contrast, the work by Erkök and Launchbury [18,19] 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 [28]) a fixed point operator that satisfies 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 1 and 2 illustrate the technique by taking an existing monadic metalanguage MML^S with ML-style references [34, Sec.3] and extending it with value recursion. Section 3 explains our technique in general terms and illustrates its robustness via three additional short examples: non-determinism, exceptions, and parallelism. Section 4 discusses in detail the important case of the continuation monad: it explains the full subtleties of value recursion in the presence of continuations and state and gives a proof of type safety for the resulting language. Section 5 recalls the equational axioms for value recursion in [18], and discusses them in our context, providing a counterexample to the left-shrinking axiom. Finally Section 6 concludes and discusses related work.

1. 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 [34], *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 irrelevant term-constructors it suffices to give local simplification rules, that can be applied non-deterministically (because they are semantic preserving). In contrast, computationally relevant term-constructors must be evaluated in the order specified by the monadic combinators. For such terms, we adopt well-established techniques for specifying the operational semantics of programming languages using evaluation contexts and abstract machine transitions (see [39]).

The syntax of MML^S is abstracted over basic types b, variables $x \in \mathsf{X}$, and locations $l \in \mathsf{L}$.

- Types $\boxed{\tau \in \mathsf{T} ::= b \mid \tau_1 \to \tau_2 \mid M\tau \mid R\tau}$

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

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

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

$$\text{set}\ \frac{\Gamma\vdash_\Sigma e_1\colon R\tau\quad \Gamma\vdash_\Sigma e_2\colon\tau}{\Gamma\vdash_\Sigma \mathit{set}\ e_1\ e_2\colon M(R\tau)}$$

Table 1. Type System for MML^S

- Terms $\boxed{\begin{array}{ll} e\in\mathsf{E}::= & x\mid\lambda x.e\mid e_1e_2\mid \mathit{ret}\ e\mid \mathit{do}\ x\leftarrow e_1;e_2\mid\\ & l\mid \mathit{new}\ e\mid \mathit{get}\ e\mid \mathit{set}\ e_1\ e_2\end{array}}$

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 with contents e, $\mathit{get}\ e$ which returns the contents of the reference e, 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 $\Gamma\colon\mathsf{X}\stackrel{fin}{\to}\mathsf{T}$ is a type assignment for variables $x\colon\tau$ and $\Sigma\colon\mathsf{L}\stackrel{fin}{\to}\mathsf{T}$ 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, *i.e.*, the compatible closure of $(\lambda x.e_2)e_1\Longrightarrow e_2\{x\colon=e_1\}$.

The computation relation $\mathit{Id}\longmapsto\mathit{Id}'\mid\mathsf{done}$ (see Table 2) is defined using the additional notions of evaluation contexts, stores and configurations $\mathit{Id}\in\mathsf{Conf}$:

- Evaluation contexts $\boxed{E\in\mathsf{EC}\ ::=\Box\mid E[\mathit{do}\ x\leftarrow\Box;e]}$
- Stores $\mu\in\mathsf{S}\stackrel{\Delta}{=}\mathsf{L}\stackrel{fin}{\to}\mathsf{E}$ map locations to their contents which are expressions (not necessarily "values").
- 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.

The first three rules are administrative: (A.1) finds the first simple command to execute recording the rest of the computation in the context; the context is popped by (A.2) when the current command returns; but if the context is empty, (A.0)

Administrative steps

$$\begin{array}{rl}
\text{(A.0)} & (\mu, ret\ e, \square) \longmapsto \mathsf{done} \\
\text{(A.1)} & (\mu, do\ x \leftarrow e_1; e_2, E) \longmapsto (\mu, e_1, E[do\ x \leftarrow \square; e_2]) \\
\text{(A.2)} & (\mu, ret\ e_1, E[do\ x \leftarrow \square; e_2]) \longmapsto (\mu, e_2\{x := e_1\}, E)
\end{array}$$

Imperative steps

$$\begin{array}{rl}
\text{(new)} & (\mu, new\ e, E) \longmapsto (\mu\{l : e\}, ret\ l, E) \text{ where } l \notin \mathsf{dom}(\mu) \\
\text{(get)} & (\mu, get\ l, E) \longmapsto (\mu, ret\ e, E) \text{ with } e = \mu(l) \\
\text{(set)} & (\mu, set\ l\ e, E) \longmapsto (\mu\{l = e\}, ret\ l, E) \text{ with } l \in \mathsf{dom}(\mu)
\end{array}$$

TABLE 2. Computation Relation for MML^S

terminates the execution. The next three rules formalize the semantics of the state-specific operations. When modifying the store, we write $\mu\{l : e\}$ for initializing a new location l with e and $\mu\{l = e\}$ for updating an existing location l with e.

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

There are alternative ways to give semantics to a programming language, but all of them should agree on basic observations of program behavior. Termination is among the most basic observations, and for MML^S it can be defined as follows.

Definition 1.1 (Termination)**.** Given a well-typed program e, *i.e.*, $\emptyset \vdash_\emptyset e : M\tau$, we say that e terminates $\overset{\Delta}{\iff} (\emptyset, e, \square) \overset{*}{\Longrightarrow} \mathsf{done}$, where $\Longrightarrow\ =\ \longrightarrow \cup \longmapsto$.

In other words, e terminates, if there is a sequence of simplification and computation steps starting from the initial configuration $(\emptyset, e, \square)$ and leading to done. Once the meaning of basic observations has been given, one can introduce a Morris's style contextual equivalence [35].

Definition 1.2 (Observational Equivalence)**.** We say that e_1 is observationally equivalent to e_2, written $e_1 \approx e_2$, if for all contexts C such that $C[e_1]$ and $C[e_2]$ are well-typed programs, we have that $C[e_1]$ terminates if and only if $C[e_2]$ terminates.

2. 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 in the next section.

- 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 $(X|\mu, e, E) \in \mathsf{Conf} \stackrel{\Delta}{=} \mathcal{P}_{\mathit{fin}}(\mathsf{X}) \times \mathsf{S} \times \mathsf{E} \times \mathsf{EC}$. 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:

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

The simplification relation is extended with the rule $\mathit{fix}\ x.e \longrightarrow e\{x{:}=\mathit{fix}\ x.e\}$ for *fix*-unfolding.

The computation relation $\mathit{Id} \longmapsto \mathit{Id}' \mid \mathsf{done} \mid \mathsf{err}$ may now raise an error and is defined by the rules in Table 2, modified to propagate the set X unchanged, and the following new rules for recursive monadic bindings:

(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 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) 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 the definition of substitution on evaluation contexts must take the captured variables into account (see Definition 4.6).

Definition 1.1 of termination (and consequently of observational equivalence) extends straightforwardly to $\mathsf{MML}^S_{\mathit{fix}}$.

Definition 2.1 (Termination)**.** Given a well-typed program e, we say that e terminates $\stackrel{\Delta}{\iff} (\emptyset|\emptyset, e, \Box) \stackrel{*}{\Longrightarrow} \mathsf{done}$.

2.1. Discussion

Divergence. In the proposed extension we have added *fix*-unfolding to simplification, thus we have endorsed the view that divergence (and general recursion) is not a computational effect. However, a *purist* approach should consider *fix*-unfolding a computation rule:

(M.0) $(X|\mu, \mathit{fix}\ x.e, E) \longmapsto (X|\mu, e\{x{:}=\mathit{fix}\ x.e\}, E)$

Types. In [18] the fixed point constructs have a slightly different typing:

- For *mfix* the bound variable is of type τ not $M\tau$: $\dfrac{\Gamma, x\colon\tau \vdash_\Sigma e\colon M\tau}{\Gamma \vdash_\Sigma \mathit{mfix}\ x.e\colon M\tau}$

  This rule allows the use of x at type τ before the recursion is resolved, as in $(\mathit{mfix}\ x.\mathit{set}\ x\ 0)\colon M(R\ \mathit{int})$. In [18] this premature attempt to use x is identified with *divergence*, while we consider it a *monadic error* (which could be prevented by more refined type systems [9,17]). The difference of typing reflects this desire and is not an intrinsic limitation of our approach.
- Similarly for *fix* the bound variable is of type τ not $M\tau$: $\dfrac{\Gamma, x\colon\tau \vdash_\Sigma e\colon \tau}{\Gamma \vdash_\Sigma \mathit{fix}\ x.e\colon \tau}$

  This typing requires recursive definitions at *all types*; we only require them at *computational types*.

Encoding *fix*. Is it necessary to have two recursive constructs $\mathit{fix}\, x.e$ and $\mathit{Mfix}\, x.e$? No, we show that *Mfix* subsumes *fix*. First, we reformulate the computation rule (M.2), so that the operational semantics of *Mfix* is given independently from *fix* (see the purity axiom in Section 5):

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

Then, we define *fix* in terms of *Mfix*

$$\mathit{fix}\ x.e \equiv \mathit{force}\ (\mathit{Mfix}\ x'.\mathit{ret}\ e\{x\colon= \mathit{force}\ x'\}) \text{ where } \mathit{force}\ e \equiv \mathit{do}\ x \leftarrow e; x$$

The definition uses *Mfix* at type $M^2\tau$ to define *fix* at type $M\tau$. Since the body of the *Mfix* definition has no effects (it is of the form $\mathit{ret}\ \ldots$), the *Mfix* computation returns immediately computing an x' of type $M^2\tau$. After the recursion is resolved, the final result and all uses of x' are *force*d to remove the outer (trivial) computation layer. Indeed, one can show that the definition of *fix* using *Mfix* is well-typed and that the computation rule (M.0) for *fix*-unfolding is *derivable*. But despite the above definability result, *fix* has a simpler semantics, given by the simplification rule for *fix*-unfolding, thus it is more natural to use *fix* whenever possible.

Recursion Variables. In the specific case of $\mathsf{MML}^S_{\mathit{fix}}$ we can simplify the operational semantics, by exploiting the following invariant (where FV and CV are sets of variables formally defined in Definition 4.6).

Lemma 2.2. *If* $(X|\mu, e, E) \longmapsto (X'|\mu', e', E')$*,* $\mathrm{FV}(\mu, e) \subseteq \mathrm{CV}(E) \subseteq X$ *and* $\mathrm{FV}(E) = \emptyset$*, then* $\mathrm{FV}(\mu', e') \subseteq \mathrm{CV}(E') \subseteq X'$ *and* $\mathrm{FV}(E') = \emptyset$*.*

In other words, since the initial configuration $(\emptyset|\emptyset, e, \square)$ has an evaluation context with no free variables, then in all reachable configurations we have that $\mathrm{FV}(E) = \emptyset$ and there is no need to consider substitution instances of E. Furthermore, the only free variables in the configuration are those currently "captured" by the evaluation context and recorded in X. Thus, when an evaluation context $\mathit{Mfix}\ x.\square$ is popped, we can remove the corresponding x from the set X, effectively treating X as a *stack* (one can go further, see [2], and identify X with $\mathrm{CV}(E)$). Summing up, in the case of $\mathsf{MML}^S_{\mathit{fix}}$, we can simplify (M.2) to:

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

However, our aim is an operational semantics that works with *arbitrary* computational effects, and an invariant like the one above does not hold in the case of continuations (Section 4).

3. General Construction with Examples

We now outline a general methodology for adding value recursion to a monadic metalanguage MML. The methodology is not a formal construction. One of the main difficulties in formalizing it is that the shape of configurations may differ greatly depending on the computational effects. In the following, the shape of configurations is general enough to handle the monads of state, non-determinism, exceptions, parallelism, and continuations.

We denote with X, E, EC and Conf the syntactic categories for variables, terms, evaluation contexts and configurations of MML, but there could be other syntactic categories. For each syntactic category C of MML, there is a *corresponding* syntactic category $\mathsf{C}_{\mathit{fix}}$ of $\mathsf{MML}_{\mathit{fix}}$, defined as follows:

- $\boxed{e \in \mathsf{E}_{\mathit{fix}}\ ::= \mathit{fix}\ x.e \mid \mathit{Mfix}\ x.e \mid \text{and the productions of } \mathsf{E}}$
- $\boxed{E \in \mathsf{EC}_{\mathit{fix}}\ ::= E[\mathit{Mfix}\ x.\Box] \mid \text{and the productions of } \mathsf{EC}}$
- $\boxed{\mathsf{C}_{\mathit{fix}}\ ::= \text{same productions of } \mathsf{C}}$ for other syntactic categories
- $(X|\mathit{Id}) \in \mathsf{Conf}_{\mathit{fix}}\ ::= (X|\ldots)$ where $X \subseteq_{\mathit{fin}} \mathsf{X}$ and $\ldots$ production for Conf

We write $\mathit{Id}[-]$ for a configuration with one hole for a thread, where a thread is represented by a pair (e, E), and $\mathit{Id}[(e, E)]$ for the configuration obtained by placing a thread (e, E) in the hole.

The simplification relation $\underset{\mathit{fix}}{\longrightarrow}$ for $\mathsf{MML}_{\mathit{fix}}$ is the compatible closure of the simplification rules for MML and *fix*-unfolding $\mathit{fix}\ x.e \longrightarrow e\{x{:=}\ \mathit{fix}\ x.e\}$.

For the computation relation $(X|\mathit{Id}) \underset{\mathit{fix}}{\longmapsto} (X'|\mathit{Id}') \mid \mathsf{done} \mid \mathsf{err}$ of $\mathsf{MML}_{\mathit{fix}}$, the extension is more involved:

- the old rules $\mathit{Id} \longmapsto \mathit{Id}' \mid \mathsf{done}$ of MML are adapted to propagate the additional components.

(old.0) $(X|\mathit{Id}) \underset{\mathit{fix}}{\longmapsto} \mathsf{done}$ if $\mathit{Id} \longmapsto \mathsf{done}$ is a computation rule of MML and similarly for other answers like done in the range of the computation relation.

(old.1) $(X|\mathit{Id}) \underset{\mathit{fix}}{\longmapsto} (X|\mathit{Id}')$ if $\mathit{Id} \longmapsto \mathit{Id}'$ is a computation rule of MML

- The introduction of a new clause for evaluation contexts for *Mfix* requires our three rules for pushing and returning from the new context:

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

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

(err) $(X|Id[(x, E)]) \underset{fix}{\longmapsto} \mathsf{err}$ where $x \in X$

- The introduction of the new clause of evaluation contexts may require additional rules depending on the effects. For example, in the case of exceptions below, the interaction between the effect of raising an exception and the new evaluation context requires an additional rule.

3.1. Non-determinism

We consider the extension of MML^S (and MML^S_{fix}) with non-deterministic choice $(e_1\ or\ e_2)$, whose typing rule 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, namely:

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

The *list* monad in Haskell can be related to the non-determinism semantics given above using the following intuition: the list (of configurations) represents the frontier of an expanding tree of reachable configurations. Therefore, instead of having two rules for the choice operator, a small-step semantics would describe how to advance the frontier

$$L_1@[(X|\mu, e_1\ or\ e_2, E)]@L_2 \longmapsto L_1@[(X|\mu, e_1, E), (X|\mu, e_2, E)]@L_2$$

This implies that each configuration in the list maintains its own store and set of recursion variables: further choices or modifications to the store done in one branch will not affect the other branches.

To help illustrate this point, consider the following term in an extension of the monadic metalanguage with constants and recursive data:

$$\begin{array}{l} \mathbf{data}\ \tau = Rec(Int, M\tau) \\ p = Mfix\ x.(ret\ Rec(1,x))\ or\ (ret\ Rec(2,x)) \end{array}$$

According to the computation rules, we push the evaluation context $Mfix\ x.\Box$ and then make a non-deterministic choice. The evaluation thus proceeds according to one of the following sequences:

$$\ldots \longmapsto (X, x|\mu, ret\ Rec(1,x), Mfix\ x.\Box) \longmapsto (X, x|\mu, ret\ Rec(1, \mathit{fix}\ x.ret\ Rec(1,x)), \Box)$$

$$\ldots \longmapsto (X, x|\mu, ret\ Rec(2,x), Mfix\ x.\Box) \longmapsto (X, x|\mu, ret\ Rec(2, \mathit{fix}\ x.ret\ Rec(2,x)), \Box)$$

where it is clear that the recursion is resolved independently for each possible choice (this separation is not apparent when one maintains all possible configurations in a list).

3.2. Exceptions

We consider the extension of MML^S (and $\mathsf{MML}^S_{\mathit{fix}}$) with the ability to raise an exception *fail*, and a construct *handle* e_1 e_2 for handling an exception raised during the evaluation of e_1. The typing rules for these two constructs are:

$$\frac{}{\Gamma \vdash_\Sigma \mathit{fail}\colon M\tau} \qquad \frac{\Gamma \vdash_\Sigma e_1\colon M\tau \quad \Gamma \vdash_\Sigma e_2\colon M\tau}{\Gamma \vdash_\Sigma \mathit{handle}\ e_1\ e_2\colon M\tau}$$

The addition of exceptions introduces a new form of termination $\mathit{Id} \longmapsto \mathsf{fail}$ for the computation relation, due to an uncaught exception. The configurations for MML^S and $\mathsf{MML}^S_{\mathit{fix}}$ are unchanged, but the BNF for evaluation contexts has an extra clause $\boxed{E \in \mathsf{EC} \mathrel{+}= E[\mathit{handle}\ \Box\ e]}$, which delimits the scope of the exception handler e. The computation relation for MML^S is extended with the following rules:

- (F.0) $(\mu, \mathit{fail}, \Box) \longmapsto \mathsf{fail}$, *i.e.*, computation terminates due to an uncaught exception
- (F.do) $(\mu, \mathit{fail}, E[\mathit{do}\ x \leftarrow \Box; e_2]) \longmapsto (\mu, \mathit{fail}, E)$, *i.e.*, an exception is propagated until it is caught by an handler
- (H.1) $(\mu, \mathit{handle}\ e_1\ e_2, E) \longmapsto (\mu, e_1, E[\mathit{handle}\ \Box\ e_2])$, *i.e.*, the handler e_2 is pushed on E
- (H.2) $(\mu, \mathit{ret}\ e_1, E[\mathit{handle}\ \Box\ e_2]) \longmapsto (\mu, \mathit{ret}\ e_1, E)$, *i.e.*, the handler e_2 is ignored
- (H.3) $(\mu, \mathit{fail}, E[\mathit{handle}\ \Box\ e_2]) \longmapsto (\mu, e_2, E)$, *i.e.*, the handler e_2 is executed

With exceptions the computation relation $\underset{\mathit{fix}}{\longmapsto}$ for $\mathsf{MML}^S_{\mathit{fix}}$ is given according to the general methodology which adapts the old rules and adds the three rules for value recursion. We also have two additional rules to deal with the additional form of termination and the interaction between fail and *Mfix*-contexts:

- (old.F) $(X|\mathit{Id}) \underset{\mathit{fix}}{\longmapsto} \mathsf{fail}$ if $\mathit{Id} \longmapsto \mathsf{fail}$ is a computation rule of MML
- (F.Mfix) $(X|\mathit{Id}[(\mathit{fail}, E[\mathit{Mfix}\ x.\Box])]) \underset{\mathit{fix}}{\longmapsto} (X|\mathit{Id}[(\mathit{fail}, E)])$, *i.e.*, we propagate the exception as in (F.do).

As a simple example, consider the following term $\mathit{Mfix}\ x.\mathsf{fail}$ which consists of a recursive computation that does not return a proper value. The evaluation proceeds as follows:

$$(X|\mu, \mathit{Mfix}\ x.\mathsf{fail}, \Box) \longmapsto (X, x|\mu, \mathsf{fail}, \mathit{Mfix}\ x.\Box) \longmapsto (X, x|\mu, \mathsf{fail}, \Box) \longmapsto \mathsf{fail}$$

This semantics is consistent with the axiomatization of Erkök and Launchbury [18] which implies $\mathit{mfix}\ x.e = e$, when x does not occur free in e.

3.3. Parallelism

We consider the extension of MML^S (and $\mathsf{MML}^S_{\mathit{fix}}$) with a construct $\mathit{spawn}\ e_1\ e_2$ to spawn a thread executing e_1 in parallel with the current thread which continues with the execution of e_2. The typing rule for spawn is:

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

The configurations for MML^S become $\langle \mu, N\rangle \in \mathsf{Conf} \stackrel{\Delta}{=} \mathsf{S} \times \mathcal{M}_{\mathit{fin}}(\mathsf{E} \times \mathsf{EC})$, *i.e.*, instead of one thread (e, E) we have a finite multi-set N of threads sharing the store μ, and the computation relation $\mathit{Id} \longmapsto \mathit{Id}' \mid \mathsf{done}$ is defined by the rules:

- Administrative steps: threads act independently

(done) $\langle \mu, \emptyset\rangle \longmapsto \mathsf{done}$ termination occurs when all threads have completed

(A.0) $\langle \mu, (\mathit{ret}\ e, \square) \uplus N\rangle \longmapsto \langle \mu, N\rangle$

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

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

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

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

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

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

- Step for spawning a new thread

(spawn) $\langle \mu, (\mathit{spawn}\ e_1\ e_2, E) \uplus N\rangle \longmapsto \langle \mu, (e_1, \square) \uplus (e_2, E) \uplus N\rangle$

Configurations for $\mathsf{MML}^S_{\mathit{fix}}$ become $\langle X|\mu, N\rangle \in \mathsf{Conf} \stackrel{\Delta}{=} \mathcal{P}_{\mathit{fin}}(\mathsf{X}) \times \mathsf{S} \times \mathcal{M}_{\mathit{fin}}(\mathsf{E} \times \mathsf{EC})$, *i.e.*, the threads in the multi-set share also the set X of recursive variables generated so far. The computation relation $\mathit{Id} \longmapsto \mathit{Id}' \mid \mathsf{done} \mid \mathsf{err}$ is defined by the rules above (modified to propagate X unchanged) and the following rules for $\mathit{Mfix}\ x.e$:

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

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

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

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

4. References and Continuations

In this section we consider in full detail the monadic metalanguage $\mathsf{MML}^{SK}_{\mathit{fix}}$, obtained from $\mathsf{MML}^S_{\mathit{fix}}$ by adding continuations (to make the technical exposition self-contained we define $\mathsf{MML}^{SK}_{\mathit{fix}}$ from scratch). This special case is very appropriate to expose the subtleties of value recursion. In particular, we outline a proof of type safety (see Section 4.2), by identifying the key technical lemmas. The type system we use is rather weak, since it cannot statically detect attempts to

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

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

$$\text{fix}\ \frac{\Gamma,x\colon M\tau\vdash_\Sigma e\colon M\tau}{\Gamma\vdash_\Sigma \mathit{fix}\ x.e\colon M\tau}\quad \text{Mfix}\ \frac{\Gamma,x\colon M\tau\vdash_\Sigma e\colon M\tau}{\Gamma\vdash_\Sigma \mathit{Mfix}\ x.e\colon M\tau}$$

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

$$\text{set}\ \frac{\Gamma\vdash_\Sigma e_1\colon R\tau\quad \Gamma\vdash_\Sigma e_2\colon\tau}{\Gamma\vdash_\Sigma \mathit{set}\ e_1\ e_2\colon M(R\tau)}$$

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

$$\text{throw}\ \frac{\Gamma\vdash_\Sigma e_1\colon K\tau\quad \Gamma\vdash_\Sigma e_2\colon M\tau}{\Gamma\vdash_\Sigma \mathit{throw}\ e_1\ e_2\colon M\tau'}$$

TABLE 3. Type System for $\mathsf{MML}^{SK}_{\mathit{fix}}$

use a recursive variable before it is resolved. However, the main purpose of this section is to serve as a template to establish similar (type safety) results for other monadic metalanguages with value recursion, and to provide the formal definitions and technical properties that have been avoided so far.

4.1. Syntax, Types, and Semantics

The syntax of $\mathsf{MML}^{SK}_{\mathit{fix}}$ 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 $\boxed{\begin{array}{ll} e\in\mathsf{E}::= & x\mid\lambda x.e\mid e_1e_2\mid \mathit{fix}\ x.e\mid\\ & \mathit{ret}\ e\mid \mathit{do}\ x\leftarrow e_1;e_2\mid \mathit{Mfix}\ x.e\mid\\ & l\mid \mathit{new}\ e\mid \mathit{get}\ e\mid \mathit{set}\ e_1\ e_2\mid\\ & k\mid \mathit{callcc}\ x.e\mid \mathit{throw}\ e_1e_2\end{array}}$

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 $\mathit{callcc}\ x.e$ binds the current continuation to x and continues with the execution of e; the execution of the expression $\mathit{throw}\ e_1e_2$ ignores the current continuation and executes e_2 in the context of the continuation e_1 instead. This effectively "jumps" to the point where the continuation e_1 was captured by callcc with e_2

as the result of the *callcc* expression. By jumping to the same continuation more than once, it is thus possible for the associated *callcc* expression to return more than once with a different value each time.

Table 3 gives the typing rules for deriving judgments of the form $\Gamma \vdash_\Sigma e\colon\tau$, where $\Gamma\colon \mathsf{X} \overset{fin}{\to} \mathsf{T}$ is a type assignment for variables $x\colon\tau$ and $\Sigma\colon \mathsf{L}\cup\mathsf{K} \overset{fin}{\to} \mathsf{T}$ 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: $(\lambda x.e_2)e_1 \longrightarrow e_2\{x\colon= e_1\}$
fix: $\mathit{fix}\ x.e \longrightarrow e\{x\colon= \mathit{fix}\ x.e\}$

Definition 4.1. The compatible closure $\xrightarrow{R}$ of a binary relation R on terms (s.t. $e\ R\ e'$ implies $\mathrm{FV}(e') \subseteq \mathrm{FV}(e)$) is given by:

$e_1 \xrightarrow{R} e_2 \overset{\Delta}{\Longleftrightarrow} e_1 \equiv C[e]$ and $e_2 \equiv C[e']$ with $e\ R\ e'$ and C context with one hole.

We write $=$ for the reflexive, symmetric and transitive closure of $\longrightarrow$. The following are desirable properties of simplification (identified in [34]), among them only Proposition 4.4 is relevant to the proof of type safety.

Proposition 4.2 (Congr)**.** *The equivalence $=$ induced by $\longrightarrow$ is a congruence.*

Proof. Clearly $=$ is an equivalence, it is a congruence, *i.e.*, $\dfrac{e_1 = e_2}{C[e_1] = C[e_2]}$, because $\longrightarrow$ is the compatible closure of a relation. ☐

Proposition 4.3 (CR)**.** *The simplification relation $\longrightarrow$ is confluent.*

Proof. The rewriting rules defining $\longrightarrow$ are left-linear and non-overlapping, thus $\longrightarrow$ is confluent by a general result on combinatory reduction systems. The proof uses an auxiliary relation $\underset{1}{\twoheadrightarrow}$, called 1-step parallel reduction, s.t. $\longrightarrow\ \subseteq\ \underset{1}{\twoheadrightarrow}\ \subseteq\ \overset{*}{\longrightarrow}$ and satisfying the diamond property. ☐

Proposition 4.4 (SR)**.** *If $\Gamma \vdash_\Sigma e\colon\tau$ and $e \longrightarrow e'$, then $\Gamma \vdash_\Sigma e'\colon\tau$.*

Proof. The proof relies on two properties of the type system: substitution and replacement. subst $\dfrac{\Gamma \vdash_\Sigma e_1\colon\tau_1 \quad \Gamma, x\colon\tau_1 \vdash_\Sigma e_2\colon\tau_2}{\Gamma \vdash_\Sigma e_2\{x\colon= e_1\}\colon\tau_2}$ allows to prove SR for the two rewriting rules defining $\longrightarrow$. While the replacement property

If $\Gamma \vdash_\Sigma C[e]\colon\tau$, then exists Γ' and τ' s.t.
$\Gamma' \vdash_\Sigma e\colon\tau'$ and $\Gamma \vdash_\Sigma C[e']\colon\tau$ whenever $\Gamma' \vdash_\Sigma e'\colon\tau'$

allows to derive SR for the compatible closure $\xrightarrow{R}$ from SR for R. ☐

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

- Evaluation contexts $\boxed{E \in \mathsf{EC}\ ::= \square \mid E[\mathit{do}\ x \leftarrow \square; e] \mid E[\mathit{Mfix}\ x.\square]}$

- 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.
- Computational redexes

$$\boxed{\begin{array}{rl} r \in \mathsf{R} ::= & \mathit{ret}\ e \mid \mathit{do}\ x \leftarrow e_1; e_2 \mid x \mid \mathit{Mfix}\ x.e \mid \\ & \mathit{new}\ e \mid \mathit{get}\ l \mid \mathit{set}\ l\ e \mid \\ & \mathit{callcc}\ x.e \mid \mathit{throw}\ k\ e \end{array}}$$

Remark 4.5. 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 [39]. In such reduction semantics there is no need to 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 [4, 5, 33]), 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 4.6. 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 $\Box\{x' := e'\} \triangleq \Box$
- $\mathrm{CV}(E[\mathit{do}\ x \leftarrow \Box; e]) \triangleq \mathrm{CV}(E)$,
  $\mathrm{FV}(E[\mathit{do}\ x \leftarrow \Box; e]) \triangleq \mathrm{FV}(E) \cup (\mathrm{FV}(e) \backslash (\mathrm{CV}(E) \cup x))$ and
  $$(E[\mathit{do}\ x \leftarrow \Box; e])\{x' := e'\} \triangleq \begin{cases} E'[\mathit{do}\ x \leftarrow \Box; e] & x' \in \mathrm{CV}(E) \\ E'[\mathit{do}\ x \leftarrow \Box; e\{x' := e'\}] & \text{otherwise} \end{cases}$$
  with $E' \equiv E\{x' := e'\}$ (the bound variable x can be renamed to be different from x' and from any of the free variables of e').
- $\mathrm{CV}(E[\mathit{Mfix}\ x.\Box]) \triangleq \mathrm{CV}(E) \cup x$, $\mathrm{FV}(E[\mathit{Mfix}\ x.\Box]) \triangleq \mathrm{FV}(E)$ and
  $(E[\mathit{Mfix}\ x.\Box])\{x' := e'\} \triangleq E'[\mathit{Mfix}\ x.\Box]$ with $E' \equiv E\{x' := e'\}$

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 variable x are replaced anywhere in the configuration)
(err)	$(X\|\mu,\rho,x,E) \longmapsto \mathsf{err}$ where $x \in X$ (attempt to use unresolved 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 $\mathsf{MML}^{SK}_{\mathit{fix}}$

The confluent simplification relation $\longrightarrow$ on terms extends in the obvious way to a confluent relation (denoted $\longrightarrow$) on stores, evaluation contexts and configurations. The following lemma establishes a useful invariant on configurations independent from typing, namely every configuration $(X|\mu,\rho,e,E)$ reachable from an initial one $(\emptyset|\emptyset,\emptyset,e_0,\Box)$ satisfies the property $\mathrm{FV}(\mu,\rho,e,E)\cup\mathrm{CV}(\rho,E)\subseteq X$. Since the property is trivially satisfied on initial configurations with all the sets empty, it suffices to show that it is preserved by simplification and computation steps.

Lemma 4.7. *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* $\mathrm{FV}(E')\subseteq\mathrm{FV}(E)$
- $\mathrm{FV}(e'_l)\subseteq\mathrm{FV}(e_l)$ *for* $e_l=\mu(l)$ *and* $e'_l=\mu'(l)$
- $\mathrm{CV}(E'_k)=\mathrm{CV}(E_k)$ *and* $\mathrm{FV}(E'_k)\subseteq\mathrm{FV}(E_k)$ *for* $E_k=\rho(k)$ *and* $E'_k=\rho'(k)$

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* $\mathrm{FV}(\mu',\rho',e',E')\cup\mathrm{CV}(\rho',E')\subseteq X'$.

Proof. The property of simplification follows from $\mathrm{FV}(e')\subseteq\mathrm{FV}(e)$ whenever $e\longrightarrow e'$, and the fact that simplification cannot modify the *shape* of a store, evaluation context or continuation environment. The property of computation is proved by considering each computation rule separately, and exploiting basic properties of substitution $\_\{x:=e\}$. We consider two cases:

(A.2) $(X|\mu,\rho,ret\ e_1,E[do\ x\leftarrow\Box;e_2])\longmapsto(X|\mu,\rho,e_2\{x:=e_1\},E)$. Since X, μ and ρ do not change, it suffices to derive $\mathrm{FV}(e_2\{x:=e_1\},E)\cup\mathrm{CV}(E)\subseteq X$ from $\mathrm{FV}(e_1,E[do\ x\leftarrow\Box;e_2])\cup\mathrm{CV}(E[do\ x\leftarrow\Box;e_2])\subseteq X$.
$\mathrm{CV}(E)\subseteq X$ because $\mathrm{CV}(E)=\mathrm{CV}(E[do\ x\leftarrow\Box;e_2])$.
$\mathrm{FV}(E)\subseteq X$ because $\mathrm{FV}(E)\subseteq\mathrm{FV}(E[do\ x\leftarrow\Box;e_2])$.
$\mathrm{FV}(e_2\{x:=e_1\})\subseteq(\mathrm{FV}(e_2)\backslash x)\cup\mathrm{FV}(e_1)\subseteq X$ because $\mathrm{FV}(e_1)\subseteq X$ and $\mathrm{FV}(e_2)\subseteq\mathrm{FV}(E[do\ x\leftarrow\Box;e_2])\cup\mathrm{CV}(E)\cup x\subseteq X\cup x$.

(M.2) $(X|\mu,\rho,ret\ e,E[Mfix\ x.\Box])\longmapsto(X|\overline{\mu},\overline{\rho},ret\ \overline{e},\overline{E})$ where $\overline{\bullet}$ stands for $\bullet\{x:=\mathit{fix}\ x.ret\ e\}$. We have to derive $\mathrm{FV}(\overline{\mu},\overline{\rho},\overline{e},\overline{E})\cup\mathrm{CV}(\overline{\rho},\overline{E})\subseteq X$ from $\mathrm{FV}(\mu,\rho,e,E)\cup\mathrm{CV}(E)\cup x\subseteq X$. We focus on e and E, since the reasoning for μ and ρ is similar.
$\mathrm{FV}(\overline{e},\overline{E})\subseteq X$ because $\mathrm{FV}(e,E)\subseteq X$ and $\mathrm{FV}(\mathit{fix}\ x.ret\ e)\subset X$.
$\mathrm{CV}(\overline{E})\subseteq X$ because $\mathrm{CV}(\overline{E})=\mathrm{CV}(E)$.

□

The following property is not relevant for type safety, but establishes a key property of simplification, namely if a computation rule is enabled in a configuration, it is still enabled if the configuration is simplified. Moreover, if the program fragment under consideration is a computational redex, further simplification does not enable more computational rules.

Theorem 4.8 (Bisim). *If $Id\equiv(X|\mu,\rho,e,E)$ 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 $\mathsf{Conf}\cup\{\mathsf{done},\mathsf{err}\}$.

Proof. An equivalent statement, but easier to prove, is obtained by replacing $\stackrel{*}{\longrightarrow}$ with 1-step parallel reduction $\underset{1}{\longrightarrow\!\!\!\!\!\rightarrow}$. A key observation for proving the bisimulation result is that simplification applied to a computational 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 4. □

4.2. Type Safety

We prove that in $\mathsf{MML}^{SK}_{\mathit{fix}}$ execution of a well-typed program $\emptyset\vdash_\emptyset e_0\colon M\tau$ does not get stuck, *i.e.*, every configuration Id reachable from the initial one $(\emptyset|\emptyset,\emptyset,e_0,\Box)$ can progress $Id\Longrightarrow$ by a simplification or computation step. Thus execution of e_0 stops either because of termination (done) or because of an attempt to use an unresolved variable (err). Following [39] we prove type safety by establishing Subject Reduction and Progress for *well-formed configurations*.

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 $M\tau$.

$$\square\ \frac{}{\Delta, \square{:}\, M\tau \vdash_\Sigma \square{:}\, M\tau}$$

$$E\text{-do}\ \frac{\Delta, \square{:}\, M\tau_2 \vdash_\Sigma E{:}\, M\tau' \quad \Delta, x{:}\,\tau_1 \vdash_\Sigma e{:}\, M\tau_2}{\Delta, \square{:}\, M\tau_1 \vdash_\Sigma E[do\ x \leftarrow \square; e]{:}\, M\tau'}$$

$$E\text{-Mfix}\ \frac{\Delta, \square{:}\, M\tau \vdash_\Sigma E{:}\, M\tau'}{\Delta, \square{:}\, M\tau \vdash_\Sigma E[\mathit{Mfix}\ x.\square]{:}\, M\tau'}\ \Delta(x) = M\tau$$

TABLE 5. Well-formed Evaluation Contexts for $\mathsf{MML}^{SK}_{\mathit{fix}}$

Definition 4.9. $\Delta \vdash_\Sigma (X|\mu, \rho, e, E){:}\,\tau' \stackrel{\Delta}{\Longleftrightarrow} \mathsf{dom}(\Sigma) = \mathsf{dom}(\mu) \uplus \mathsf{dom}(\rho)$, $\mathsf{dom}(\Delta) = X$ and

- $\Delta \vdash_\Sigma e{:}\, M\tau$ and $\Delta, \square{:}\, M\tau \vdash_\Sigma E{:}\, M\tau'$ are derivable (see Table 5) for some type τ
- $\Delta \vdash_\Sigma e_l{:}\,\tau_l$ is derivable when $e_l = \mu(l)$ and $R\tau_l = \Sigma(l)$
- $\Delta, \square{:}\, M\tau_k \vdash_\Sigma E_k{:}\, M\tau'$ is derivable when $E_k = \rho(k)$ and $K\tau_k = \Sigma(k)$

The formation rules of Table 5 for deriving $\Delta, \square{:}\, M\tau \vdash_\Sigma E{:}\, M\tau'$ ensure that Δ assigns a computational type to all captured variables of E. Moreover, we have the following substitution lemma for evaluation contexts.

Lemma 4.10. *The following rule is admissible*

$$E\text{-subst}\ \frac{\Delta \vdash_\Sigma e_0{:}\, M\tau_0 \quad \Delta, x_0{:}\, M\tau_0, \square{:}\, M\tau \vdash_\Sigma E{:}\, M\tau'}{\Delta, x_0{:}\, M\tau_0, \square{:}\, M\tau \vdash_\Sigma E\{x_0{:}= e_0\}{:}\, M\tau'}$$

Proof. By induction on the derivation of $\Delta, x_0{:}\, M\tau_0, \square{:}\, M\tau \vdash_\Sigma E{:}\, M\tau'$.

We write $\overline{\bullet}$ for $\bullet\{x_0{:}= e_0\}$. The only interesting case is the (E-do) rule, when we have to derive $\Delta, x_0{:}\, M\tau_0, \square{:}\, M\tau_1 \vdash_\Sigma \overline{E[do\ x \leftarrow \square; e]}{:}\, M\tau'$ from both $\Delta, x_0{:}\, M\tau_0, \square{:}\, M\tau_2 \vdash_\Sigma E{:}\, M\tau'$ and $\Delta, x_0{:}\, M\tau_0, x{:}\,\tau_1 \vdash_\Sigma e{:}\, M\tau_2$.

By IH we have $\Delta, x_0{:}\, M\tau_0, \square{:}\, M\tau_2 \vdash_\Sigma \overline{E}{:}\, M\tau'$. If $x_0 \in \mathrm{CV}(E)$, then we have that $\overline{E[do\ x \leftarrow \square; e]} \equiv \overline{E}[do\ x \leftarrow \square; e]$, and the conclusion is immediate. Otherwise $\overline{E[do\ x \leftarrow \square; e]} \equiv \overline{E}[do\ x \leftarrow \square; \overline{e}]$, thus we need $\Delta, x_0{:}\, M\tau_0, x{:}\,\tau_1 \vdash_\Sigma \overline{e}{:}\, M\tau_2$, which follows from substitution and weakening for the type system. $\square$

We can now formulate the SR and progress properties for $\mathsf{MML}^{SK}_{\mathit{fix}}$.

Theorem 4.11 (SR)**.**

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

Proof. The first claim is an easy consequence of Proposition 4.4. The second is proved by case-analysis on the computation rules of Table 4. The most interesting case is (M.2). We know $\Delta \vdash_\Sigma (X|\mu, \rho, ret\ e, E[\mathit{Mfix}\ x.\square]){:}\,\tau'$ and we want to derive $\Delta \vdash_\Sigma (X|\overline{\mu}, \overline{\rho}, ret\ \overline{e}, \overline{E}){:}\,\tau'$ where $\overline{\bullet}$ stands for $\bullet\{x{:}= \mathit{fix}\ x.ret\ e\}$.

Let τ be s.t. $\Delta \vdash_\Sigma \mathit{ret}\ e\colon M\tau$ and $\Delta, \Box\colon M\tau \vdash_\Sigma E[M\mathit{fix}\ x.\Box]\colon M\tau'$, thus we must have $\Delta(x) = M\tau$ and $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$. By (fix) and weakening we derive $\Delta \vdash_\Sigma \mathit{fix}\ x.\mathit{ret}\ e\colon M\tau$. We can now prove $\Delta \vdash_\Sigma (X|\overline{\mu}, \overline{\rho}, \mathit{ret}\ \overline{e}, \overline{E})\colon \tau'$.

- $\Delta \vdash_\Sigma \mathit{ret}\ \overline{e}\colon M\tau$ by substitution and weakening
- $\Delta, \Box\colon M\tau \vdash_\Sigma \overline{E}\colon M\tau'$ by Lemma 4.10

the properties of $\overline{\mu}$ and $\overline{\rho}$ (in Definition 4.9) are proved by similar arguments. $\Box$

Lemma 4.12. *If $\Delta \vdash_\Sigma e\colon \tau'$ and e is a $\longrightarrow$-normal form, then*

- *$\tau' \equiv M\tau$ implies e is a computational redex*
- *$\tau' \equiv (\tau_1 \to \tau_2)$ implies e is a λ-abstraction*
- *$\tau' \equiv R\tau$ implies e is a location l*
- *$\tau' \equiv K\tau$ implies e is a continuation k*

Proof. By induction on e. The only cases that use the IH are: $e_1 e_2$, $\mathit{get}\ e$, $\mathit{set}\ e_1\ e_2$ and $\mathit{throw}\ e_1\ e_2$.

$e_1 e_2$: if $\Delta \vdash_\Sigma e_1 e_2\colon \tau_2$ and in normal form, the we must have (for some τ_1) $\Delta \vdash_\Sigma e_1\colon \tau_1 \to \tau_2$ and in normal form. Thus e_1 must be a λ-abstraction (by IH for e_1), and $e_1 e_2$ is a β-redex. This contradict the assumption that $e_1 e_2$ is in normal form.

***get* e:** if $\Delta \vdash_\Sigma \mathit{get}\ e\colon \tau'$ and in normal form, then we must have (for some τ) $\tau' = M\tau$ and $\Delta \vdash_\Sigma e\colon R\tau$ and in normal form. Thus e must be a location l (by IH for e), and $\mathit{get}\ e$ is a computational redex.

The cases $\mathit{set}\ e_1\ e_2$ and $\mathit{throw}\ e_1\ e_2$ are similar to $\mathit{get}\ e$. $\Box$

In a well-typed configuration there must be a way to make progress either by making a simplification step or a computational step. In the latter case the computational step might lead to another configuration, to done indicating proper termination, or to err indicating an attempt to use an unresolved recursion variable.

Theorem 4.13 (Progress)**.** *If $\Delta \vdash_\Sigma (X|\mu, \rho, e, E)\colon \tau'$, then*

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

Proof. We have (for some τ) that $\Delta \vdash_\Sigma e\colon M\tau$. When $e \in \mathsf{R}$ we show that $(X|\mu, \rho, e, E) \longmapsto$ by case analysis on the structure of computational redexes.

***ret* e:** rules (A.0), (A.2) or (M.2) are applicable, depending on E

***do* $x \leftarrow e_1; e_2$:** rule (A.1) is applicable

x: rule (err) is applicable, because $x \in \mathsf{dom}(\Delta) = X$ by well-formedness of the configuration

***get* l:** rule (get) is applicable, because $l \in \mathsf{dom}(\Sigma) = \mathsf{dom}(\mu)$ by the well-formedness of the configuration

The other cases are similar to either $\mathit{do}\ x \leftarrow e_1; e_2$ or $\mathit{get}\ l$. When $e \notin \mathsf{R}$, then e cannot be a $\longrightarrow$-normal form, because of Lemma 4.12. $\Box$

Given subject reduction and progress, we thus have type safety for the monadic metalanguage with continuations and state.

Corollary 4.14. *If* $\emptyset \vdash_\emptyset e_0 \colon M\tau$ *and* $(\emptyset|\emptyset, \emptyset, e_0, \Box) \xRightarrow{*} Id$*, then* $Id \Longrightarrow$ *.*

Continuations and state are so expressive that they can define or simulate many other monads [21]. But the type safety result does not immediately carry over to other monads. For example, in situations involving communication and parallelism, type safety need to be formulated differently, since a well-typed program may deadlock.

5. Axioms for Value Recursion

We discuss two of the main axioms for defining value recursion in [18], claming validity for one of them and providing a counterexample for the other.

5.1. Purity

The *purity* axiom:

$$\mathit{mfix}\ x.\mathit{ret}\ e = \mathit{ret}\ (\mathit{fix}\ x.e)$$

is the property which ensures that *mfix* coincides with *fix* for pure computations. In our case, because of the differences in typing between *mfix* and *Mfix*, this corresponding axiom would be that $\mathit{Mfix}\ x.\mathit{ret}\ e = \mathit{fix}\ x.\mathit{ret}\ e$.

In an operational setting the way to validate an equational axiom $e_1 = e_2$ is to prove that it is observationally valid, *i.e.*, $e_1 \approx e_2$. A standard strategy for proving $e_1 \approx e_2$ is the *bisimulation proof technique*. In our setting it amounts to find a binary relation R on Conf with the following properties:

(1) $(\emptyset|\emptyset, \emptyset, C[e_1], \Box) R (\emptyset|\emptyset, \emptyset, C[e_2], \Box)$ for any program (closing) context C

(2) $Id_1 R Id_2$ and $Id_1 \Longrightarrow D_1$ imply $Id_2 \xRightarrow{*} D_2$ and $D_1 \tilde{R} D_2$ for some D_2

(3) $Id_1 R Id_2$ and $Id_2 \Longrightarrow D_2$ imply $Id_1 \xRightarrow{*} D_1$ and $D_1 \tilde{R} D_2$ for some D_1

where $\Longrightarrow\ =\ \longrightarrow \cup \longmapsto$ and $\tilde{R}$ extends R to $\mathsf{Conf} \mid \mathsf{done} \mid \mathsf{err}$, namely $\tilde{R} = R \cup \{(\mathsf{done}, \mathsf{done}), (\mathsf{err}, \mathsf{err})\}$.

For instance, to prove that $e_1 \approx e_2$ when $e_1 \longrightarrow e_2$ one could use the following relation $Id_1 R Id_2 \overset{\Delta}{\Longleftrightarrow} Id_1 \xrightarrow{*} Id_2$. To check that R has the required properties we have to exploit Theorems 4.3 and 4.8.

Indeed it is possible to prove the purity axiom using our semantics.

Proposition 5.1 (Purity)**.** $\mathit{Mfix}\ x.\mathit{ret}\ e \approx \mathit{fix}\ x.\mathit{ret}\ e$

5.2. Left-Shrinking

The *left-shrinking* axiom:

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

states that computations which do not refer to the recursive variable can be moved outside the recursive definition.

The axiom is known to be satisfied in many monads [18]. However a related equivalence is known to be incorrect in Scheme due to the presence of *callcc* [7]. It was argued [18] 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 [13,22]. We provide a complete counterexample to this axiom to illustrate the extreme subtlety and complications of using value recursion with general effects which include continuations, and to suggest that it is unlikely that value recursion can satisfy a large class of "interesting" axioms in the general case.

The example (inspired by examples by Bawden [7] and Carlsson [13]) is written in a Haskell-like extension of $\mathsf{MML}^{SK}_{\mathit{fix}}$ with booleans, pairs, etc. (but *Mfix* is not a legal Haskell identifier, as it starts with an uppercase letter). We will arrange for the final result of our example to be a recursive pair whose first component is an *Int* and whose second component is a computation producing another recursive pair as formalized by the type $RP\ m$ below (and where m is the monadic type constructor). The basic idea in the counterexample is to use continuations to execute the same computation more than once returning a different value each time. This requires a recursive type like $V\ m$ below (where again m is the monadic type constructor):

```
data RP m = RP (m (Int, RP m))
data V m  = Ret (K (RP m)) | Jump (K (V m))
```

Values of type $V\ m$ are continuations which either accept the final value or which jump back and restart the computation again with a new continuation.

The following two terms should be equal by left-shrinking:

```
lhs :: Monad m ⇒ m (Int, RP m)
lhs = Mfix (\ x →
             do p ← callcc (\ k → return (Jump k))
                case p of
                 Jump k → do v ← callcc (\ c → throw k (return (Ret c)))
                             return (1, v)
                 Ret c → throw c (return (RP x)))
```

```
rhs :: Monad m ⇒ m (Int, RP m)
rhs = do p ← callcc (\ k → return (Jump k))
         Mfix (\ x →
               case p of
                Jump k → do v ← callcc (\ c → throw k (return (Ret c)))
                            return (1, v)
                Ret c → throw c (return (RP x)))
```

In our semantics (extended with simplification rules for booleans, pairs, etc) the two terms evaluate differently. In the *lhs*, the evaluation enters the *Mfix* expression and *then* continuations are captured and invoked within the body of the *Mfix* expression. Eventually the body of *Mfix* evaluates to ($return\,(1,\ RP\ x)$) where x is the recursive variable, and the whole expression produces the well-defined value $(1,\ RP\ (fix\ x.\ return\ (1,\ RP\ x)))$.

In more detail, the evaluation of the *lhs* consists of the following macro steps:

- x: recursive variable x is created.
- k: continuation k is bound to the evaluation context
  $E_k\ =\ Mfix\ (\backslash\ x\ \rightarrow\ \mathbf{do}\ p\ \leftarrow\ \Box;\ e)$ where

$$
\begin{array}{ll}
e\ =\ \mathbf{case}\ p\ \mathbf{of} & \\
\quad Jump\ k\ \rightarrow\ \mathbf{do}\ v\ \leftarrow\ callcc\ (\backslash\ c\ \rightarrow\ throw\ k\ (return\ (Ret\ c))) & \\
\qquad\qquad\qquad\qquad return\ (1, v) & \\
\quad Ret\ c\ \rightarrow\ throw\ c\ (return\ (RP\ x)) &
\end{array}
$$

- c: branch *Jump k* is selected, and continuation c is bound to
  $E_c\ =\ Mfix\ (\backslash\ x\ \rightarrow\ \mathbf{do}\ v\ \leftarrow\ \Box;\ return\ (1, v))$
- *throw k*: $return\ (Ret\ c)$ is thrown to E_k, thus branch *Ret c* is selected.
- *throw c*: $return\ (RP\ x)$ is thrown to E_c which produces
  $Mfix\ (\backslash\ x\ \rightarrow\ \mathbf{do}\ v\ \leftarrow\ return\ (RP\ x)\ ;\ return\ (1, v))$
- mfix: the evaluation of the body of *Mfix* returns
  $Mfix\ (\backslash\ x\ \rightarrow\ return\ (1, (RP\ x)))$
- resolve x: the final answer is $return\ (1, (RP\ fix\ x.\ (1, (RP\ x))))$

In the case of the *rhs*, a continuation is captured *before* we ever start evaluating the *Mfix* expression. Then p gets bound to the value (*Jump k*) and the *Mfix* expression is entered a first time with recursive variable x. This however immediately jumps back and rebinds p to a new value (*Ret c*) where c is a continuation which refers to the captured recursive variable x. But jumping back and rebinding p to (*Ret c*) starts a new evaluation of the *Mfix* expression with recursive variable x' this time. This evaluation invokes c with the value ($RP\ x'$) which results in the body of the original evaluation of *Mfix* (with recursive variable x) to produce ($return\ (1,\ RP\ x')$). Thus the *rhs* produces the value $(1,\ RP\ x')$ which has a free unresolved recursive variable.

In more detail the evaluation consists of the following steps:

- k: continuation k is bound the evaluation context
  $E'_k = \mathbf{do}\ p\ \leftarrow\ \Box;\ Mfix\ (\backslash\ x\ \rightarrow\ e)$ where e is the term in macro step k in the evaluation of the *lhs*.
- x: recursive variable x is created.
- c: branch *Jump k* is selected, and continuation c is bound to the evaluation context E_c introduced in macro step c in the evaluation of the *lhs*.
- *throw k*: $return\ (Ret\ c)$ is thrown to E'_k, which binds p to *Ret c*.
- x': recursive variable x', different from x, is created.
- *throw c*: branch *Ret c* is selected, and $return\ (RP\ x')$ is thrown to E_c which produces
  $Mfix\ (\backslash\ x\ \rightarrow\ \mathbf{do}\ v\ \leftarrow\ return\ (RP\ x')\ ;\ return\ (1, v))$

- mfix: the evaluation of the body of *Mfix* returns $\mathit{Mfix}\,(\backslash\; x \;\rightarrow\; \mathit{return}\,(1,\; \mathit{RP}\; x'))$
- resolve x: the final answer is $\mathit{return}\,(1,\; \mathit{RP}\; x')$, with x' unresolved.

5.3. Scheme semantics

Our semantics also differs from the Scheme semantics. The semantics of a **letrec** expression in Scheme is given as follows [30]:

$$\begin{aligned}(\mathbf{letrec}\;((x\;e))\;e') = \;&(\mathbf{let}\;((x\;(\mathit{void})))\\ &\quad(\mathbf{let}\;((v\;e))\\ &\quad\quad(\mathbf{begin}\;(\mathbf{set!}\;x\;v)\;e')))\end{aligned}$$

To understand the definition, one should note the nature of variables in Scheme: a variable declaration implicitly allocates a location and a variable use implicitly reads the corresponding location. This is unlike our presentation so far in which variables always refer to values and locations are explicitly created and read with *new* and *get*. Given this understanding, the meaning of a **letrec** expression is as follows:

- Allocate a location and initialize it to an unusable value.
- Evaluate the right hand side e; this evaluation cannot read the contents of the location, *i.e.*, it cannot use x in an evaluation context position.
- Update the location with the resulting value and evaluate the body.

The differences between our approach and the Scheme semantics become more evident if we attempt to define a fixed point operator that captures the Scheme semantics: one might be tempted to use the following definition:

$$(\mathit{schemeFix}\;f) \;=\; (\mathbf{letrec}\;((x\;(f\;x)))\;x)$$

but unfortunately the application $(f\;x)$ attempts to prematurely read the location associated with x.

Indeed the presence of locations and assignments cannot be encapsulated in the definition of **letrec**: a judicious use of *callcc* exposes them and can be used to define first-class reference cells [7,22].

6. Conclusion and Related Work

Recursion is a pervasive feature in many general purpose programming languages. Cook [14] has demonstrated its importance in clarifying the semantics of object-oriented languages. Bracha [12] has abstracted from the object-oriented paradigm and proposed mixins and related operations as a way for supporting modularity in a variety of programming languages. Ancona and Zucca [1,3] have proposed *CMS* as a foundational calculus for mixins more suitable for theoretical studies, although able to express (in term of few primitives) the variety of operations identified by Bracha. All these works show that recursion, and more specifically the management of mutually recursive definitions, is important also for programming in the large.

This paper investigates the interaction of recursion with computational effects. We have worked in the setting monadic metalanguages, since they provide a general framework for computational effects, and have proposed a uniform way of adding *value recursion* to a monadic metalanguage equipped with an *abstract* operational semantics described by a simplification and computation relation.

Other researchers have looked (or are looking) at value recursion at a different level of abstraction or with a different focus. We attempt a classification of research topics that are related to value recursion, and mention some of the most significant contributions (without any attempt to be exhaustive).

- Value Recursion and Monads [18,19]. While using Haskell for modeling hardware circuits and stream-oriented computations, Launchbury, Lewis, and Cook realized that the monadic infrastructure could not be used in describing recursive circuits [32]. This observation led Erkök and Launchbury to abstract from the specific application domain, investigate the semantics of value recursion, and implement it as an extension to Haskell. Their work is also prominent for taking an axiomatic approach and providing equational reasoning principles for value recursion. Paterson [36] provides similar results in the more general framework of arrows [27].
- Axiomatic/Categorical Treatments of Recursion. The semantics of value recursion has been considered also in a categorical setting, in particular [8] has adapted the notion of *trace* to *premonoidal* categories [37], which include the Kleisli category for a strong monad.
- Implementations of Value Recursion. For specific constrained notions of computations, for example for state, I/O, and several other monads, it is possible to implement value recursion in terms of the usual fixed point construct. For example, given a simple state monad defined by the constructor: $Ma = Int \to (a, Int)$, one can define:

  $$Mfix\ x.e = \lambda s.(fix\ p.(\lambda x.e)\ (ret\ (fst\ p))\ s)$$

  which is how current Haskell implementations [23,29] deal with value recursion for the state and IO monads (replacing our *Mfix* with *mfix*). In the general setting where the notion of computation is not restricted, all known implementations of value recursion rely on a variant of the SECD update-in-place trick [16,30,31]. This implementation strategy can often be optimized, but sometimes with some imposed restrictions [11,17,26,38].
- Value Recursion and Objects. The semantics of objects proposed by Cook [15] identifies a class c with a function $f\colon R \to R$ (from records to records); a *new* object o of class c is (the record) obtained by taking the fixed point of f. When objects have a state, some computations (initialization) have to be done once, at object-creation time. If computations during initialization are heavily constrained, one could model a class as

a computation of a function $f'\colon M(R \to R)$, but in general a more subtle form of recursion is needed [9], namely value recursion.

- Value Recursion and Modules [2, 3, 17, 24, 25]. The importance of supporting recursion at the module level has been observed by Bracha, who proposed the notion of mixin. Even for rich module languages, like ML-modules, there is a need for extensions supporting inter-module recursion. One of the stumbling blocks in designing such extensions is the interaction of module-level recursion and core-level computational effects. Value recursion offers (in combination with standard recursion) an enhanced control of this interaction.
- Type Systems for Value Recursion [10, 17, 24, 25]. Value recursion introduces a new form of error, caused by an attempt to use a recursive variable before it is bound to a value (we call such a variable *unresolved*). It is desirable to have type systems (or program analyses) allowing static detection of such errors.

Acknowledgments

We would like to thank Levent Erkök and Magnus Carlsson for very fruitful discussions and comments. We would also like to thank Sungwoo Park for pointing out an error in one of the examples in an earlier version of the paper. Finally the comments and questions by the referees were quite helpful in improving the paper and its presentation.

References

[1] Ancona, D. *Modular Formal Frameworks for Module Systems.* PhD thesis, Univ. di Pisa, 1998.

[2] Ancona, D., Fagorzi, S., Moggi, E., and Zucca, E. Mixin modules and computational effects. In *Proc. 30th Int'l Coll. Automata, Languages, and Programming* (2003), vol. 2719 of *LNCS*, Springer-Verlag.

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

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

[5] Ariola, Z. M., Maraist, J., Odersky, M., Felleisen, M., and Wadler, P. 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* (New York, NY, USA, 1995), ACM Press, pp. 233–246.

[6] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed. North-Holland, 1984.

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

[8] Benton, N., and Hyland, M. Traced pre-monoidal categories. In *Fixed Points in Computer Science* (July 20–21 2002), Z. Ésik and A. Ingólfsdóttir, Eds., vol. NS-02-2 of *BRICS Notes Series*, pp. 12–19.

[9] Boudol, G. The recursive record semantics of objects revisited. *Lecture Notes in Computer Science 2028* (2001), 269–283.

[10] BOUDOL, G. The recursive record semantics of objects revisited. *J. Funct. Programming* (2002). To appear.

[11] BOUDOL, G., AND ZIMMER, P. Recursion in the call-by-value λ-calculus. In *Fixed Points in Computer Science, BRICS Notes Series NS-02-2* (2002).

[12] BRACHA, G. *The Programming Language Jigsaw: Mixins, Modularity, and Multiple Inheritance.* PhD thesis, Univ. of Utah, Mar. 1992.

[13] CARLSSON, M. Value recursion in the continuation monad. Unpublished Note, Jan. 2003.

[14] COOK, W. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[15] COOK, W., AND PALSBERG, J. A denotational semantics of inheritance and its correctness. In *Conf. on Object-Oriented Programming:Systems, Languages and Applications* (1989), ACM.

[16] COUSINEAU, G., CURIEN, P. L., AND MAUNY, M. The categorical abstract machine. In *Functional Programming Languages and Computer Architecture* (Sept. 1985), J.-P. Jouannaud, Ed., vol. 201 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 50–64.

[17] DREYER, D., HARPER, R., AND CRARY, K. A type system for well-founded recursion. Tech. Rep. CMU-CS-03-163, Carnegie Mellon University, 2003.

[18] ERKÖK, L. *Value Recursion in Monadic Computations.* PhD thesis, OGI School of Science and Engineering, OHSU, Portland, Oregon, 2002.

[19] ERKÖK, L., AND LAUNCHBURY, J. Recursive monadic bindings. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)* (N.Y., Sept. 18–21 2000), vol. 35.9 of *ACM Sigplan Notices*, ACM Press, pp. 174–185.

[20] ERKÖK, L., LAUNCHBURY, J., AND MORAN, A. Semantics of value recursion for monadic input/output. *Journal of Theoretical Informatics and Applications 36*, 2 (2002), 155–180.

[21] FILINSKI, A. Representing monads. In *Conf. Record of 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'94, Portland, OR, USA, 17–21 Jan. 1994.* ACM Press, New York, 1994, pp. 446–457.

[22] FRIEDMAN, D. P., AND SABRY, A. Recursion is a computational effect. Tech. Rep. 546, Computer Science Department, Indiana University, Dec. 2000.

[23] GHC TEAM, T. The glasgow haskell compiler user's guide, version 4.08. Available online from `http://haskell.org/ghc/`. Viewed on 12/28/2000.

[24] HIRSCHOWITZ, T. *Mixin Modules, Modules and Extended Value Binding in a Call-By-Value Setting.* PhD thesis, Univ. Paris 7, 2003. To appear.

[25] HIRSCHOWITZ, T., AND LEROY, X. Mixin modules in a call-by-value setting. In *Programming Languages & Systems, 11th European Symp. Programming* (2002), vol. 2305 of *LNCS*, Springer-Verlag, pp. 6–20.

[26] HIRSCHOWITZ, T., LEROY, X., AND WELLS, J. B. Compilation of extended recursion in call-by-value functional languages. In *Proc. 5th Int'l Conf. Principles & Practice Declarative Programming* (2003).

[27] HUGHES, J. Generalising monads to arrows. *Sci. Comput. Program. 37*, 1-3 (2000), 67–111.

[28] Report on the programming language Haskell 98, Feb. 1999.

[29] JONES, M. P., AND PETERSON, J. C. Hugs 1.4 User Manual. Research Report YALEU/DCS/RR-1123, Yale University, Department of Computer Science, 1997.

[30] KELSEY, R., CLINGER, W., AND (EDITORS), J. R. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (Sept. 1998), 26–76.

[31] LANDIN, P. J. The mechanical evaluation of expressions. *The Computer Journal 6*, 4 (Jan. 1964), 308–320.

[32] LAUNCHBURY, J., LEWIS, J. R., AND COOK, B. On embedding a microarchitectural design language within Haskell. In *Proc. 1999 Int'l Conf. Functional Programming* (1999), ACM Press, pp. 60–69.

[33] MARAIST, J., ODERSKY, M., AND WADLER, P. The call-by-need lambda calculus. *Journal of Functional Programming 8*, 3 (May 1998), 275–317.

[34] MOGGI, E., AND FAGORZI, S. A monadic multi-stage metalanguage. In *Proc. FoSSaCS '03* (2003), vol. 2620 of *LNCS*, Springer-Verlag.

[35] Morris, J. H. *Lambda-Calculus Method of Programming Language*. PhD thesis, MIT, Dec. 1968.

[36] Paterson, R. A new notation for arrows. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming* (2001), ACM Press, pp. 229–240.

[37] Power, J., and Robinson, E. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science 7*, 5 (1997), 453–468.

[38] Waddell, O., Sarkar, D., and Dybvig, R. K. Robust and effective transformation of letrec. In *Scheme Workshop* (Oct. 2002).

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

Contents

Communicated by (The editor will be set by the publisher).
(The dates will be set by the publisher).