Semantics of *fixIO*

Levent Erkök* John Launchbury* Andrew Moran†

*OGI School of Science and Engineering, OHSU
†Galois Connections, Inc.

Abstract

Recent work on recursion over the values of monadic actions resulted in the introduction of a family of fixed point operators, one for each different kind of monadic effect. In the context of Haskell, the function *fixIO* is the corresponding operator for the IO monad. Unfortunately, both the IO monad and *fixIO* are language primitives in Haskell, i.e. they can not be defined within the language itself. Therefore, any attempt to formally reason about *fixIO* is futile without a precise semantics for computations in the IO monad. Quite recently, Peyton Jones introduced an operational semantics based on observable transitions as a method for reasoning about I/O in Haskell. Building on this work, we show how one can model *fixIO* as well, and we argue that it indeed belongs to the family of fixed point operators that enable monadic value recursion.

1 Introduction

Ever since Peyton Jones and Wadler showed how monads can be used to model I/O in a language with non-strict semantics, monadic I/O became the standard way of dealing with input/output in Haskell [13]. Together with the IO monad, a rather mysterious function called *fixIO* was also introduced. Intuitively, *fixIO* allows us to model computations that depend on *"results that are not yet computed but lazily available"* [1, section 4.1]. The functionality provided by *fixIO* is similar to that of *fixST* associated with the state monad [7].

Later work on recursion resulting from the values of monadic actions tried to explain the behavior of such fixed point operators from an axiomatic point of view [3]. It was noted that a generic fixed point operator, one that would work regardless of the underlying effect, was not available. Instead, one has to specify a "suitable" fixed point operator for each different kind of monadic effect. The meaning of "suitable" was encoded in a number of properties. The same work also provided a catalogue of operators for various monads satisfying these properties and conjectured that *fixIO* was the required operator for the IO monad in Haskell, pending a detailed treatment.

Appears in the *Fixed Points in Computer Science Workshop (FICS'01)*, a satellite workshop to PLI'2001. Sept. 7-8, 2001, Florence, Italy.

To explore the conjecture further, we need to understand *fixIO*. First of all, we need a semantics for the computations in the IO monad. Recent work by Peyton Jones introduced a semantics based on observable transitions [11], in the spirit of monadic transition systems that were previously studied by Gordon [4]. In such a system, an IO computation is viewed as a sequence of labeled transitions. Each label indicates an effect observable in the real world, similar to those in process calculi [9]. Peyton Jones' work used an embedding of a denotational semantics for the functional layer into the IO layer. However, it bypassed the details of this embedding. Such an approach is fine, as long as one is interested in the big picture. If, on the other hand, one wants to reason about *fixIO*, it becomes necessary to be explicit about the relationship between the IO and functional layers. Our aim in this paper is to bridge this gap.

Our semantics is structured in two layers: IO and functional. The semantics for the IO layer is based on the approach taken by Peyton Jones [11]. The semantics for the functional layer is based on the natural semantics for lazy evaluation of Launchbury [5]. A final set of rules precisely shows how these two layers interact with each other. It is this interaction that allows us to give a semantics for *fixIO*. The present paper contains our initial results, including a complete operational semantics for both layers. A more rigorous study of the underlying proof system is still pending.

The remainder of this paper is structured as follows. First, we motivate the usage of *fixIO* with some simple examples. Then we present a language with monadic I/O constructs, together with a two layer semantics. Then, we show how one can use the semantics to reason about programs involving *fixIO*. This is followed by a brief review of monadic value recursion and a discussion of the properties satisfied by *fixIO*. We conclude with a summary of our results and pointers for future work.

2 Motivating Examples

To get an idea of what *fixIO* does, consider the following Haskell expression of type $IO\ [Char]$:

$$\begin{aligned} &\mathit{fixIO}\ (\lambda cs.\ \mathbf{do}\ c \leftarrow \mathit{getChar} \\ &\qquad\qquad\qquad\quad \mathit{return}\ (c : cs)) \end{aligned}$$

Intuitively, when we run this computation, we expect a character to be read from the standard input, say `a`. Once the input operation is completed, the computation immediately terminates with the delivery of an infinite list of `a`'s. We will be able to pull out as many characters as we wish out of this list, following the demand-driven evaluation policy of Haskell. There are two crucial points:

- The action *getChar* is executed only once,
- The computation immediately terminates after the reading is done, i.e. the infinite list is not constructed prior to its demand. In other words, the fact that the IO monad is strict in actions but not in values is preserved by *fixIO*.

Here, we also get a feel for what *fixIO* provides: it lets us name the results of computations that will only be available later on. That is, we were able to name the result of the computation as cs, before we had its value computed. In this sense, the semantics is similar to the pure expression:

$$\begin{array}{l}\textbf{let } cs = \text{'}\mathbf{a}\text{'} : cs \\ \textbf{in } cs\end{array}$$

except, to determine the character in the list we first perform an effect via the call to *getChar*.

One other facility provided by the IO monad in Haskell is mutable variables. Here is an example showing the interaction of *fixIO* with mutable cells:

$$\begin{array}{l} \mathit{fixIO}\ (\lambda\,\tilde{}\,(x,\ \_).\ \textbf{do}\ y \leftarrow \mathit{newIORef}\ x \\ \qquad\qquad\qquad\qquad\quad \mathit{return}\ (1{:}x,\ y)) \\ \quad \gg\!= \ \lambda(\_,\ l).\ \mathit{readIORef}\ l \end{array}$$

In this example, we allocate a cell in which we store the value of the variable x, before we know what this value really is. The contents of x is determined through the fixed point computation, to be the infinite list of 1's. The call to *fixIO* returns the value (which is discarded) and the address of the cell that stores this cyclic structure. Outside of the call to *fixIO*, we dereference the address, to get back the lazily computed list of 1's. Although this example might look superficial, this is exactly what's going on whenever we have a cyclic structure with mutable nodes. For instance, we have previously described how such a technique can be used to implement doubly-linked-lists, where each node held a mutable boolean value [3]. A similar situation arises in object oriented programming, when several objects need to refer to each other cyclically [10].

Once we describe the semantics for *fixIO*, we will revisit these examples to see how our system works in practice.

3 The language

In this section, we define a language with monadic IO primitives, based on Haskell [12].

Sorts:

$$\begin{array}{rcl} c & \in & \mathit{constructors} \\ x,y,z,w & \in & \mathit{heap\ variables} \\ r,s,t & \in & \mathit{mutable\ variables} \end{array}$$

We syntactically distinguish between heap and mutable variables: they come from different alphabets.

Terms:

$$\begin{array}{rcl} M,N & ::= & x \\ & | & V \\ & | & M\ N \\ & | & \textbf{let}\ \vec{x} = \vec{M}\ \textbf{in}\ N \\ & | & \textbf{case}\ M\ \textbf{of}\ \{c_i\ \vec{x}_i \rightarrow N_i\} \end{array}$$

Values:

$$\begin{array}{rl} V ::= & c\ x_1\ x_2\ \ldots\ x_i \\ | & \lambda x.\ M \\ | & \mathit{return}\ M \mid M \gg\!= N \\ | & \mathit{getChar} \mid \mathit{putChar}\ k \\ | & \mathit{fixIO}\ M \mid \mathit{update}_z\ M \\ | & r \mid \mathit{newIORef}\ M \\ | & \mathit{readIORef}\ r \mid \mathit{writeIORef}\ r\ M \end{array}$$

Terms and values are defined mutually recursively. Programs in our language are well typed terms of type $IO\ a$, for some type a,[1] and results of programs are values, with the exception of the special value update_z. This value, associated with a heap variable z, can not appear in a valid program. Furthermore, it is never the result of any program either. It is only used internally, in giving a semantics to *fixIO*. We will explain the role of *update* in detail later. All other constructs have the same meaning as they do in Haskell. Note that IO actions are values as far as the purely functional world is concerned.

The first alternative in the definition of V covers constructors. A constructor application is treated just like a normal function application. A constructor c of arity i should be considered as an abbreviation for $\lambda x_1 \ldots x_i.\ c\ x_1 \ldots x_i$. When a constructor is fully applied, it becomes a value of its own. This is captured in the first alternative in the definition of V, where c is assumed to have arity i. Such a value has variables, rather than arbitrary terms, as the arguments to the constructor. Our rules will ensure that this is always the case. We model constants as nullary constructors, i.e. numbers, characters etc. are treated as constructors with zero arity. (As a notational hint, we use the letter k to refer to constants.)

It is worth noting that the grammar we gave describes the syntax for the reduced terms of our language rather than its concrete syntax. This distinction shows up mainly in the applications of *putChar, readIORef* and *writeIORef*. In the concrete syntax, we allow them to receive arbitrary terms as their first arguments, as in $\mathit{putChar}\ (f\,x)$. As we will see, our rules will reduce them appropriately to conform to our grammar. We will also allow ourselves to use the do-notation of Haskell, whose translation to the core syntax is trivial [12].

Execution Contexts:

$$\mathbb{E} ::= [.] \mid \mathbb{E} \gg\!= M$$

An execution context is a term with one hole.[2] We use contexts to guide our semantics. At each step, we match the current term under consideration against this grammar, such that there is a rule that corresponds to the term filling the hole. In practice, this simply amounts to looking at the leftmost branch in the tree of $\gg\!=$ nodes.

Heaps: A heap is a finite partial function from heap variables to terms extended with a special blackhole value:

$$\Gamma, \Delta, \Theta :: x \rightharpoonup M \cup \{\bullet\}$$

[1] For the purposes of this paper, we completely ignore the issue of types. We assume that the usual Haskell rules apply to determine well typed terms, and we never try to evaluate ill-typed terms. Typing of Haskell programs has been discussed in detail in the literature [12].

[2] Other authors use the term *evaluation context* for this concept. We prefer the term execution, since such a context can only only be filled by an IO action in this case, which is going to be executed next.

A blackhole binding, written as $z \mapsto \bullet$, indicates that z is known but its binding is not accessible. Notice that $\bullet$ is a detectable bottom, and is not directly available to the user (much like $update_z$). Its role will be explained in detail later.

Program state: A running program is identified by its heap and its term state given by:

$$\begin{array}{rcll} P, Q, R & ::= & M & \text{Current term} \\ & | & \langle x \rangle_r & \text{An } \textit{IORef} \text{ named } r \\ & | & P \mid Q & \text{Parallel composition} \\ & | & \nu r.P & \text{Restriction} \end{array}$$

We use the notation

$$\Gamma : P$$

to capture a program in execution. A program state $\Gamma : P$ is called *closed*, if Γ contains bindings for all the free variables appearing in the term state P.

Notice that the term state allows parallel composition of terms, as in $M \mid N$. This generality is needed when one extends the system to handle concurrency primitives [11]. However, we do not use this generality here: For us, a term state is a single term and a number of passive containers that hold references to heap variables.

We need some machinery in formalizing our arguments. First, we need to be able to tell free and bound names in certain contexts. Our free-name function fn can take a heap, a context or a term state as an argument. For a context or a term state, fn simply returns the set of all free variables in it, that is any occurrence of a heap or a mutable variable that is *not* in the scope of a λ or a ν. For a heap Γ, it is defined as $fn(\Gamma) = \bigcup\{fn(M) \mid x \mapsto M \in \Gamma\}$. We treat fn as a variable arity function to simplify the notation: $fn(A, B)$ means $fn(A) \cup fn(B)$. Similarly, the function bn takes a heap and returns all the variables bound in it, i.e. $bn(\Gamma) = \{x \mid x \mapsto M \in \Gamma\}$. The notation Γ/P stands for the slice of a heap Γ, with respect to a term state P. Intuitively, it is the subset of Γ that is reachable from the free names of P. More precisely, for a given Γ and P, let

$$\begin{array}{rcl} S_0 & = & fn(P) \\ S_{i+1} & = & S_i \cup (\bigcup\{fn(M) \mid x \in S_i \wedge x \mapsto M \in \Gamma\}) \end{array}$$

and let $S = \bigcup_{i \in \mathbb{N}} S_i$. Then,

$$\Gamma/P = \{x \mapsto M \mid x \in S, x \mapsto M \in \Gamma\}$$

The final bit of notation we use is $(\Gamma, x \mapsto M)$, which stands for $\Gamma \cup \{x \mapsto M\}$, with the side condition $x \notin bn(\Gamma)$. It simply means that we extend the heap Γ with the binding $x \mapsto M$, where x is a fresh variable.

4 Semantics

We describe the semantics of our language in two layers. The IO layer takes care of the interaction with the outer world and manages mutable variables. The functional layer handles pure computations. A final set of rules regulate the interaction between these two layers.

4.1 IO layer

Figure 1 gives the transition rules for the IO layer. A transition labeled $!c$ means that the character c is printed on standard output, and one labeled $?c$ means that a character is read from standard input. Although our rules are similar to those given by Peyton Jones [11], the following points are worth mentioning:

- As in the natural semantics of Launchbury [5], we keep track of a separate global heap to store values of variables,
- Unlike the awkward squad paper, our reference cells only store heap variables, rather than arbitrary terms. This is necessary in order to model sharing implied by lazy evaluation.

The *FIXIO* rule is modeled after knot tying recursion semantics. We first create a new heap variable, called z, whose value is not yet known. This is achieved by binding it to $\bullet$. Then, we call the function and pass it the argument z, and proceed normally. If the evaluation of this function needs to know the value of z, the derivation will get stuck with a detected blackhole. Otherwise, z could be passed around, stored in data structures etc: Notice that it is just a normal heap variable. Once the function call completes, we update the heap variable z by the result of the function, effectively tying the knot by an application of the *UPDATE* rule. This behavior achieves the recursion implied by *fixIO*. In summary, z holds the value of the entire computation, which might in turn depend lazily on its own value, exactly the mechanism provided by *fixIO*.

4.2 Functional layer

The semantics of the functional layer closely follows Launchbury's natural semantics [5]. Some minor differences are worth mentioning:

- We introduce a new blackhole binding,
- The *APP* rule is generalized to application of terms to terms, rather than terms to just variables. Correspondingly, we do not need to perform the normalization pass,
- We perform renaming in the *LET* rule, rather than the *VAR* rule,

In the *LET* rule, we rename all bound variables $x_1 \dots x_n$ to $\hat{x}_1 \dots \hat{x}_n$ so that there won't be any name clashes in the heap when we do the additions. Similarly, the term $\hat{M}_i$ denotes the same term as M_i where each occurrence of x_i is replaced by $\hat{x}_i$. (And $\hat{N}$ as well.) The *VAR* rule is not applicable if the corresponding variable is bound to $\bullet$ in the heap. (Recall that $\bullet \notin M$.) If this is ever the case, the derivation will simply terminate with failure. This corresponds to a detected blackhole.

4.3 The marriage

Given separate semantics for the IO and functional layers, we need to specify exactly how they interact. There are two different kinds of interaction. First, whenever we try to reduce a term of the form, say, *putChar* M, we first need to consult the functional layer to reduce the term M to a character. The IO layer will then perform the output. We need similar rules for *readIORef* and *writeIORef* as well. The first three rules in Figure 3 take care of this interaction. The second kind of interaction embeds the functional world

$$
\begin{array}{rcll}
\mathbb{E}[putChar\ \ c] & \xrightarrow{!c} & \mathbb{E}[return\ ()] & (PUTC) \\
\mathbb{E}[getChar] & \xrightarrow{?c} & \mathbb{E}[return\ \ c] & (GETC)
\end{array}
$$

$$
\mathbb{E}[return\ N \ggg= \ M] \longrightarrow \mathbb{E}[M\ \ N] \qquad (LUNIT)
$$

$$
\frac{r \notin fn(\mathbb{E}, M)}{\Gamma : \mathbb{E}[newIORef\ \ M] \longrightarrow (\Gamma, x \mapsto M) : \nu r.(\mathbb{E}[return\ \ r] \mid \langle x \rangle_r)} \quad (NEWIO)
$$

$$
\begin{array}{rcll}
\mathbb{E}[readIORef\ \ r] \mid \langle x \rangle_r & \longrightarrow & \mathbb{E}[return\ \ x] \mid \langle x \rangle_r & (READIO) \\
\Gamma : \mathbb{E}[writeIORef\ \ r\ \ N] \mid \langle x \rangle_r & \longrightarrow & (\Gamma, y \mapsto N) : \mathbb{E}[return\ \ ()] \mid \langle y \rangle_r & (WRITEIO)
\end{array}
$$

$$
\begin{array}{rcll}
\Gamma : \mathbb{E}[fixIO\ \ M] & \longrightarrow & (\Gamma, z \mapsto \bullet) : \mathbb{E}[M\ \ z \ggg= \ update_z] & (FIXIO) \\
(\Gamma, z \mapsto \bullet) : \mathbb{E}[update_z\ \ M] & \longrightarrow & (\Gamma, z \mapsto M) : \mathbb{E}[return\ \ z] & (UPDATE)
\end{array}
$$

Figure 1: Semantics: IO layer

$$
\Gamma : V \Downarrow \Gamma : V \quad (VALUE)
$$

$$
\frac{\Gamma : M \Downarrow \Delta : \lambda y.M' \qquad (\Delta, w \mapsto N) : M'[w/y] \Downarrow \Theta : V}{\Gamma : MN \Downarrow \Theta : V} \quad (APP)
$$

$$
\frac{(\Gamma, x \mapsto \bullet) : M \Downarrow (\Delta, x \mapsto \bullet) : V}{(\Gamma, x \mapsto M) : x \Downarrow (\Delta, x \mapsto V) : V} \quad (VAR)
$$

$$
\frac{(\Gamma, \hat{x}_1 \mapsto \hat{M}_1 \cdots \hat{x}_n \mapsto \hat{M}_n) : \hat{N} \Downarrow \Delta : V}{\Gamma : \textbf{let}\ \ x_1 = M_1 \cdots x_n = M_n\ \ \textbf{in}\ \ N \Downarrow \Delta : V} \quad (LET)
$$

$$
\frac{\Gamma : M \Downarrow \Delta : c_k\ \vec{x}_k \qquad \Delta : M_k[\vec{x}_k/\vec{y}_k] \Downarrow \Theta : V}{\Gamma : \textbf{case}\ M\ \textbf{of}\ \{c_i\ \vec{y}_i \to M_i\} \Downarrow \Theta : V} \quad (CASE)
$$

Figure 2: Semantics: Functional layer

into the IO world, as modeled by the last rule in the figure. Whenever the IO layer has a functional expression to reduce (such as an application or a let binding), it uses the functional layer to do the job. In all these rules, M is assumed to be a non-value: The functional layer is consulted to reduce M to a value. As we will see later, this side condition ensures that a derivation will never loop forever due to repeated applications of the *VALUE* rule of the functional layer.

4.4 Structural rules

Finally, we need a set of structural rules to shape our proof trees. We need the following rules from the awkward squad paper: (Here, the label α ranges over empty transitions as well.)

$$
\begin{array}{rcll}
P \mid Q & \equiv & Q \mid P & (COMM) \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & (ASSOC) \\
\nu r.\nu s.P & \equiv & \nu s.\nu r.P & (SWAP)
\end{array}
$$

$$
\frac{P \xrightarrow{\alpha} Q}{P \ \mid \ R \xrightarrow{\alpha} Q \ \mid \ R} \quad (PAR)
$$

$$
\frac{P \equiv P' \qquad P' \xrightarrow{\alpha} Q' \qquad Q' \equiv Q}{P \xrightarrow{\alpha} Q} \quad (EQUIV)
$$

$$
\frac{P \xrightarrow{\alpha} Q}{\nu r.P \xrightarrow{\alpha} \nu r.Q} \quad (NU)
$$

We also get the *ALPHA* and *EXTRUDE* rules, with slight modifications. First, we need to take care of renaming in the heap, and we must make sure that we do not create a dangling reference:

$$
\frac{s \notin fn(P)}{\Gamma : \nu r.P \ \equiv \ \Gamma[s/r] : \nu s.P[s/r]} \quad (ALPHA)
$$

$$
\frac{r \notin fn(Q, \Gamma/Q)}{\Gamma : (\nu r.P) \mid Q \ \equiv \ \Gamma : \nu r.(P \mid Q)} \quad (EXTRUDE)
$$

Notice that we don't need a side condition of the form $s \notin bn(\Gamma)$ in the *ALPHA* rule. In fact, such a condition is not even well-typed: Recall that heap and mutable variables come from different alphabets, and heap only ever binds heap variables. But, in the extrude rule, it can be the case that $r \in fn(\Gamma/Q)$, that is the bound terms in the heap can contain the names of mutable variables. Section 5.3 contains an example demonstrating the situation.

We also need the following new structural rules that regulate the interaction of the heap with other bits of the semantics:

$$
\frac{P \xrightarrow{\alpha} Q}{\Gamma : P \xrightarrow{\alpha} \Gamma : Q} \quad (HEAPIN)
$$

$$
\frac{\Gamma : P \xrightarrow{\alpha} \Delta : Q}{\Gamma : P \mid R \xrightarrow{\alpha} \Delta : Q \mid R} \quad (HEAPPAR)
$$

$$
\frac{\Gamma : P \xrightarrow{\alpha} \Delta : Q}{\Gamma : \nu r.P \xrightarrow{\alpha} \Delta : \nu r.Q} \quad (HEAPNU)
$$

$$\frac{\Gamma : M \Downarrow \Delta : k}{\Gamma : \mathbb{E}[\mathit{putChar}\ M] \longrightarrow \Delta : \mathbb{E}[\mathit{putChar}\ k]} \quad (\mathit{PUTCEVAL})$$

$$\frac{\Gamma : M \Downarrow \Delta : r}{\Gamma : \mathbb{E}[\mathit{readIORef}\ M] \longrightarrow \Delta : \mathbb{E}[\mathit{readIORef}\ r]} \quad (\mathit{READIOEVAL})$$

$$\frac{\Gamma : M \Downarrow \Delta : r}{\Gamma : \mathbb{E}[\mathit{writeIORef}\ M\ N] \longrightarrow \Delta : \mathbb{E}[\mathit{writeIORef}\ r\ N]} \quad (\mathit{WRITEIOEVAL})$$

$$\frac{\Gamma : M \Downarrow \Delta : V}{\Gamma : \mathbb{E}[M] \longrightarrow \Delta : \mathbb{E}[V]} \quad (\mathit{FUN})$$

Figure 3: Semantics: Marriage of layers. All these rules are subject to the side condition that M is not a value.

$$\frac{\Gamma : P \xrightarrow{\alpha} \Delta : Q \qquad m \in M \cup \{\bullet\}}{(\Gamma, x \mapsto m) : P \xrightarrow{\alpha} (\Delta, m \mapsto s) : Q} \quad (\mathit{HEAPEXT})$$

The first three rules show how we can introduce a heap, a parallel composition or a restriction in an ongoing derivation. These rules let us to concentrate on pieces of proofs separately at first, and put them together later on as necessary. The last rule shows how to add extra bindings to a heap. This rule is useful when we take an ongoing proof and want to work on a particular term in it in isolation. In such a case, we need not carry all the bindings in the heap with us, but rather consider only those that matter for that particular term. When we are done with the subproof, we add the bindings back again, and resume normally. We will see an application of this rule in Section 6.3.

4.5 Derivations

A derivation for a program represented by the term m starts with the program state

$$\{\} : m$$

that is, with the empty heap. (Notice that the type of m is $IO\ a$ for some type a, since m represents a program.) At each step, we match the program state to one of the rules and apply it. Depending on the outcome of this process, we classify (well typed) terms into two categories:

Definition 1 (*Kinds of terms.*) Let $\Gamma : \mathbb{E}[m]$ be a closed program state. The term m is called normal if the derivation starting at this state terminates, and divergent otherwise.

A derivation can diverge by getting stuck, i.e. when we end up with program state with no applicable rules, or by being infinite, i.e. when we never run out of rules to apply. We also find the following definitions helpful:

Definition 2 (*Silent derivations.*) A derivation is silent if it contains no labeled transitions.

Definition 3 (*Strict divergence.*) A derivation is strictly divergent if it is silent and divergent.

In monadic value recursion, we deal with functions of the type $a \to IO\ a$, and we need to be able to identify when such a function is strict. That is, we need to identify the least element of $IO\ a$, for some type a. We use the following definition to serve this purpose:

Definition 4 (*Bottom of IO.*) A term m of type $IO\ a$ has the denotation $\bot$, iff the derivation for $\Gamma : m$ is strictly divergent for all Γ.

Hence, the following expression:

$$\begin{array}{l} \textbf{let}\ \mathit{loop} = \textbf{do}\ \mathit{putStr}\ \text{“}\texttt{hello}\text{”} \\ \qquad\qquad\qquad\ \ \mathit{loop} \\ \textbf{in}\ \ \mathit{loop} \end{array}$$

is *not* associated with $\bot$: Its derivation is divergent but not silent. Now we can state what it means to be a strict function of type $a \to IO\ a$:

Definition 5 (*Strict functions.*) A function f of type $a \to IO\ a$ is strict if, for all Γ, the derivation for

$$(\Gamma, x \mapsto \bullet) : f\ \ x$$

is strictly divergent.

The semantics we have described is deterministic. Given an IO computation, its derivation in our system is directly guided by its structure. Furthermore, at each step there is at most one rule that is applicable in the IO layer. The following lemma states an expected property of derivations in our system:

Lemma 1 *(Derivations for normal terms.)* Let $\Gamma : \mathbb{E}[m]$ be a closed program state where m is a normal term. The derivation starting at $\Gamma : \mathbb{E}[m]$ will take the form:

$$\Gamma : \mathbb{E}[m] \xrightarrow{\alpha} \Delta : \nu\vec{r}.(\mathbb{E}[\mathit{return}\ \ n] \mid C)$$

where n is a term, α is a (possibly empty) sequence of labels, and C is a number of (possibly zero) passive containers. The restrictions $\nu\vec{r}$ cover all the newly created references corresponding to the containers in C, if any.

Proof Since m is normal, we are guaranteed by definition 1 that the derivation will terminate. To establish the result, we simply note that the only program state that is not matched by any of the rules in our semantics is of the form $\Delta : \mathbb{E}[\mathit{return}\ \ n]$, for some term n. Since the derivation for $\Gamma : \mathbb{E}[m]$ is guaranteed to terminate, it should do so with a term of the required form. (In the course of reaching this point, passive containers might be created by the *WRITEIO* rule, and labels by the *PUTC* and *GETC* rules. The structural rule *EXTRUDE* can be used to pull all restrictions to the top level, possibly after renaming them using the *ALPHA* rule.) $\square$

5 Examples

In this section, we revisit the examples given in Section 2, and show how our semantics can handle them. For further examples of derivations, see the awkward squad paper [11]. In these examples, we will use the letters $a, b, \ldots$ to represent heap variables as well. To save space, we apply the structural rules silently and collapse a sequence of reductions in the functional layer to a single step.

5.1 An example with `getChar`

We first consider the example with *getChar*. We first remove the *do* notation in favor of explicit $\gg\!=$'s:

$$fixIO\ (\lambda cs.\ getChar \gg\!= \ \lambda c.\ return\ (c : cs))$$

We have:

$$\begin{array}{l}
\quad \{\} : fixIO\ (\lambda cs.\ getChar \gg\!= \ \lambda c.\ return\ (c : cs)) \\
\longrightarrow (\text{FIXIO - FUN}) \\
\quad \{z \mapsto \bullet,\ a \mapsto z\} : \\
\quad\quad getChar \gg\!= \ \lambda c.\ return\ (c : a) \gg\!= \ update_z \\
\xrightarrow{?ch} (\text{GETC}) \\
\quad \{z \mapsto \bullet,\ a \mapsto z\} : \\
\quad\quad return\ ch \gg\!= \ \lambda c.\ return\ (c : a) \gg\!= \ update_z \\
\longrightarrow (\text{LUNIT - FUN}) \\
\quad \{z \mapsto \bullet,\ a \mapsto z,\ b \mapsto ch\} : \\
\quad\quad return\ (b : a) \gg\!= \ update_z \\
\longrightarrow (\text{LUNIT}) \\
\quad \{z \mapsto \bullet,\ a \mapsto z,\ b \mapsto ch\} : update_z\ (b : a) \\
\longrightarrow (\text{UPDATE}) \\
\quad \{z \mapsto b : a,\ a \mapsto z,\ b \mapsto ch\} : return\ z
\end{array}$$

The derivation successfully terminates at this point, as no further rules apply. Notice that the heap now contains the cyclic structure that represents the infinite list of *ch*'s: The character that was read by the call to *getChar*. In case elements of this list is demanded in a context, the usual demand-driven rules modeled by our semantics would let us produce enough elements to satisfy the need.

5.2 Using references

We now consider the example with reference cells. Again, removing *do*-notation and simplifying the patterns to match our language, we have:

$$\begin{array}{l}
fixIO\ (\lambda t.\ newIORef\ (fst\ t) \gg\!= \ \lambda y. \\
\qquad\qquad return\ (1{:}fst\ t,\ y)) \\
\quad \gg\!= \ \lambda u.\ readIORef\ (snd\ u)
\end{array}$$

We'll first consider the *fixIO* call. To save space, we will abbreviate *newIORef* to *new* and *readIORef* to *read* :

$$\begin{array}{l}
\quad \{\} : fixIO\ (\lambda t.\ new\ (fst\ t) \gg\!= \ \lambda y. \\
\qquad\qquad return\ (1{:}fst\ t,\ y)) \\
\longrightarrow (\text{FIXIO - FUN}) \\
\quad \{z \mapsto \bullet,\ a \mapsto z\} : \\
\quad\quad new\ (fst\ a) \gg\!= \lambda y.\ return\ (1{:}fst\ a,\ y) \\
\longrightarrow (\text{NEWIO}) \\
\quad \{z \mapsto \bullet,\ a \mapsto z,\ b \mapsto fst\ a\} : \\
\quad\quad \nu r.(return\ r \gg\!= \lambda y.return\ (1{:}fst\ a,\ y) \\
\qquad\quad \gg\!= update_z \mid \langle b \rangle_r) \\
\longrightarrow (\text{LUNIT - FUN}) \\
\quad \{z \mapsto \bullet,\ a \mapsto z,\ b \mapsto fst\ a,\ c \mapsto r\} : \\
\quad\quad \nu r.(return\ (1{:}fst\ a,\ c) \gg\!= \ update_z \mid \langle b \rangle_r) \\
\longrightarrow (\text{LUNIT - UPDATE}) \\
\quad \{z \mapsto (1{:}fst\ a,\ c),\ a \mapsto z,\ b \mapsto fst\ a,\ c \mapsto r\} : \\
\quad\quad \nu r.(return\ z \mid \langle b \rangle_r)
\end{array}$$

When we consider the original expression, it is not hard to see that we will end up with:

$$\begin{array}{l}
\longrightarrow (\text{LUNIT - FUN}) \\
\quad \{z \mapsto (1 : fst\, a,\ c),\ a \mapsto z,\ b \mapsto fst\, a,\ c \mapsto r, \\
\qquad d \mapsto z\} : \ \ \nu r.(read\ (snd\ d) \mid \langle b \rangle_r) \\
\longrightarrow (\text{READIOEVAL}) \\
\quad \{z \mapsto (e, f),\ a \mapsto z,\ b \mapsto fst\, a,\ c \mapsto r,\ d \mapsto (e, f) \\
\qquad e \mapsto 1 : fst\, a, f \mapsto r\} : \ \ \nu r.(read\ r \mid \langle b \rangle_r) \\
\longrightarrow (\text{READIOREF}) \\
\quad \{z \mapsto (e, f),\ a \mapsto z,\ b \mapsto fst\, a,\ c \mapsto r,\ d \mapsto (e, f) \\
\qquad e \mapsto 1 : fst\, a, f \mapsto r\} : \ \ \nu r.(return\ b \mid \langle b \rangle_r)
\end{array}$$

Now, if we chase the value of b in the heap, we see that we will end up with a cyclic structure effectively representing the infinite lists of 1's, as intended. The most interesting step in this derivation is the application of the *READIOEVAL* rule. The function *snd* is a short hand for **case** over the pairing constructor. The *VAR* rule in the functional layer arranges for sharing, resulting in an abundance of variables in the resulting heap. Notice that, abusing the notation slightly, in the above derivation $(1 : fst\, a, c)$ refers to a function application: the pairing constructor applied to the terms $1 : fst$ and c. In the last two lines, however, (e, f) is a value, i.e. in this case, the pairing constructor applied to the right number of arguments.

5.3 Dangling references

In this section, we will consider an example demonstrating the importance of the side condition of the *EXTRUDE* rule. Consider:

$$\begin{array}{l}
\textbf{do}\ j \leftarrow new\ 5 \\
\quad\ \ k \leftarrow new\ j \\
\quad\ \ l \leftarrow read\ k \\
\quad\ \ read\ l
\end{array}$$

Or, simply:

$$new\ 5 \gg\!= \ new \gg\!= \ read \gg\!= \ read$$

We will try to give a derivation for this expression, ignoring the side condition of the *EXTRUDE* rule:

$$\begin{array}{l}
\quad \{\} : new\ 5 \gg\!= \ new \gg\!= \ read \gg\!= \ read \\
\longrightarrow (\text{NEWIOREF}) \\
\quad \{x \mapsto 5\} : \\
\quad\quad \nu j.(return\ j \gg\!= \ new \gg\!= \ read \gg\!= \ read \mid \langle x \rangle_j) \\
\longrightarrow (\text{LUNIT-NEWIOREF}) \\
\quad \{x \mapsto 5,\ y \mapsto j\} : \\
\quad\quad \nu j.(\nu k.(return\ k \gg\!= \ read \gg\!= \ read \mid \langle y \rangle_k) \mid \langle x \rangle_j) \\
\longrightarrow (\text{COMM}) \\
\quad \{x \mapsto 5,\ y \mapsto j\} : \\
\quad\quad \nu j.(\langle x \rangle_j \mid \nu k.(return\ k \gg\!= \ read \gg\!= \ read \mid \langle y \rangle_k)) \\
\longrightarrow (\text{EXTRUDE} - \textit{incorrect application}) \\
\quad \{x \mapsto 5,\ y \mapsto j\} : \\
\quad\quad \nu j.(\langle x \rangle_j) \mid \nu k.(return\ k \gg\!= \ read \gg\!= \ read \mid \langle y \rangle_k) \\
\longrightarrow (\text{LUNIT - READ - LUNIT}) \\
\quad \{x \mapsto 5,\ y \mapsto j\} : \nu j.(\langle x \rangle_j) \mid \nu k.(read\ y \mid \langle y \rangle_k) \\
\longrightarrow (\text{READIOEVAL}) \\
\quad \{x \mapsto 5,\ y \mapsto j\} : \nu j.(\langle x \rangle_j) \mid \nu k.(read\ j \mid \langle y \rangle_k)
\end{array}$$

And now, we are stuck! The mutable variable j is not visible at this point. Since we were not careful in applying the extrude rule, we have created a dangling reference. Let's construct the slice when we apply the extrude rule:

$$S_0 = \{y\},\ S_1 = \{y, j\},\ S_2 = S_1 = S_\infty$$

Therefore, the slice is: $\{y \mapsto j\}$. Since $j \in fn(\{y \mapsto j\})$, extrude is not applicable. The side condition prevents the creation of the dangling reference.

6 Monadic value recursion

Equipped with the semantics we have presented so far, we are now in a position to look at monadic value recursion in Haskell's IO monad. But first, we briefly review monadic value recursion in general.

Monadic value recursion aims at explaining the behavior of recursion under the presence of effects modeled by monads. According to the usual unfolding view of recursion, one "unfolds" the body of a recursive definition as many times as necessary. If the underlying computation involves effects, unfolding will repeat these effects. This is precisely the required behavior, for instance, when we model recursively defined functions that might perform side effects: Each time the function calls itself recursively, we want the effect to take place again. For a certain class of problems, however, this view of recursion is not appropriate. In these cases, recursion needs to be performed only over the values, without repeating (or losing) the associated effects. The unfolding view of recursion fails to differentiate between values and effects, simply repeating them both until a fixed point is reached. (A premier example of the use of monadic value recursion is in modeling circuits using monads [6].) We use the term "value recursion" for this concept.

Not surprisingly, this sort of recursion is modeled by "monadic value recursion operators". We claim that *fixIO* is such an operator for the IO monad in Haskell. In our earlier work, we have identified several properties that monadic value recursion operators are expected to satisfy [3]. Based on the semantics we have described so far, we now investigate these properties in detail for *fixIO*.

6.1 Strictness

Strictness property states that, if $f :: a \to IO\ a$ is a strict function, *fixIO f* is $\perp$. That is:

$$f\ \perp = \perp\ \to \mathit{fixIO}\ f\ = \perp$$

Assuming f is a strict function (definition 5), we have:

$$\begin{array}{l} \quad \Gamma : \mathit{fixIO}\ f \\ \longrightarrow \text{(FIXIO)} \\ \quad (\Gamma,\ z \mapsto \bullet) : f\ z \gg\!= \ \mathit{update}_z \end{array}$$

The current context specifies that the application $f\ z$ should be evaluated. But since f is strict, by definition 5 the derivation will strictly diverge. But then, by definition 4, this implies that *fixIO f* is $\perp$.

To illustrate the notion of strictness for IO computations, we consider some examples: Using **if** as a shorthand for **case** over the boolean type and abbreviating *return* to *ret*, consider:

$$\begin{array}{l} \quad \{\} : \mathit{fixIO}\ (\lambda x.\mathbf{if}\ x\ ==\ 0\ \mathbf{then}\ \mathit{ret}\ 1\ \mathbf{else}\ \mathit{ret}\ 2) \\ \longrightarrow \text{(FIXIO - FUN)} \\ \quad \{z \mapsto \bullet,\ a \mapsto z\} : \\ \qquad \mathbf{if}\ a\ =\ 0\ \mathbf{then}\ \mathit{ret}\ 1\ \mathbf{else}\ \mathit{ret}\ 2 \gg\!= \mathit{update}_z \\ \longrightarrow \text{(FUN)} \\ \ldots\ \textit{detected blackhole}\ \ldots \end{array}$$

On the other hand, consider the non-strict function:

$$\lambda x.\ \mathit{return}\ x :: \mathit{Char} \to \mathit{IO}\ \mathit{Char}$$

Notice that it returns a computation successfully. Of course, if the result of the fixed-point computation is used, it will still diverge, but for a different reason. Consider:

$$\begin{array}{l} \quad \{\} : \mathit{fixIO}\ (\lambda x.\ \mathit{ret}\ x) \gg\!= \ \mathit{putChar} \\ \longrightarrow \text{(FIXIO - FUN)} \\ \quad \{z \mapsto \bullet,\ a \mapsto z\} : \mathit{ret}\ a \gg\!= \ \mathit{update}_z \gg\!= \ \mathit{putChar} \\ \longrightarrow \text{(LUNIT)} \\ \quad \{z \mapsto \bullet,\ a \mapsto z\} : \mathit{update}_z\ a \gg\!= \ \mathit{putChar} \\ \longrightarrow \text{(UPDATE)} \\ \quad \{z \mapsto a,\ a \mapsto z\} : \mathit{ret}\ z \gg\!= \ \mathit{putChar} \\ \longrightarrow \text{(LUNIT)} \\ \quad \{z \mapsto a,\ a \mapsto z\} : \mathit{putChar}\ z \\ \longrightarrow \text{(PUTCEVAL)} \\ \ldots\ \textit{detected blackhole}\ \ldots \end{array}$$

The last step diverges, since the *VAR* rule will get stuck trying to reduce z to a character.

Similarly, the function:

$$\lambda a.\ \mathit{putChar}\ \text{'s'} \gg \mathbf{if}\ a\ \mathbf{then}\ \mathit{ret}\ 1\ \mathbf{else}\ \mathit{ret}\ 2$$

is not strict either. Here is the derivation (abbreviating *putChar* to *put*):

$$\begin{array}{l} \quad \{\} : \\ \qquad \mathit{fixIO}\ (\lambda a.\ \mathit{put}\ \text{'s'} \gg \mathbf{if}\ a\ \mathbf{then}\ \mathit{ret}\ 1\ \mathbf{else}\ \mathit{ret}\ 2) \\ \longrightarrow \text{(FIXIO - FUN)} \\ \quad \{z \mapsto \bullet,\ a \mapsto z\} : \\ \qquad \mathit{put}\ \text{'s'} \gg \mathbf{if}\ a\ \mathbf{then}\ \mathit{ret}\ 1\ \mathbf{else}\ \mathit{ret}\ 2 \\ \qquad\quad \gg\!= \ \mathit{update}_z \\ \xrightarrow{!\mathtt{s}} \text{(PUTC)} \\ \quad \{z \mapsto \bullet,\ a \mapsto z\} : \\ \qquad \mathbf{if}\ a\ \mathbf{then}\ \mathit{ret}\ 1\ \mathbf{else}\ \mathit{ret}\ 2 \gg\!= \ \mathit{update}_z \\ \longrightarrow \text{(FUN)} \\ \ldots\ \textit{detected blackhole}\ \ldots \end{array}$$

But, before getting stuck, we see the character `s` printed, which is the correct behavior in this case.

6.2 Purity

Purity property states that, for all $f :: a \to a$:

$$\mathit{fixIO}\ (\mathit{return}\ .\ f)\ =\ \mathit{return}\ (\mathit{fix}\ f)$$

On the lhs, we have:

$$\begin{array}{l} \quad \Gamma : \mathit{fixIO}\ (\mathit{return}\ .\ h) \\ \longrightarrow \text{(FIXIO - FUN)} \\ \quad (\Gamma,\ z \mapsto \bullet,\ a \mapsto z) : (\mathit{return}\ .\ h)\ a \gg\!= \ \mathit{update}_z \\ \longrightarrow \text{(LUNIT)} \\ \quad (\Gamma,\ z \mapsto \bullet,\ a \mapsto z) : \mathit{update}_z\ (h\ a) \\ \longrightarrow \text{(UPDATE)} \\ \quad (\Gamma,\ z \mapsto h\ a,\ a \mapsto z)\ : \mathit{return}\ z \end{array}$$

Consider the right-hand-side, with

$$\mathit{fix}\ f\ =\ \mathbf{let}\ x\ =\ f\ x\ \mathbf{in}\ x$$

We get:

Γ : *return* (*fix* h)

To show that these two program states are observationally equivalent, it suffices to argue that the term z with respect to the heap $\{\Gamma, z \mapsto h\ a, a \mapsto z\}$, and the term *fix* h with respect to the heap Γ can not be distinguished by the functional layer. A simple proof in the functional layer shows that

$$\Gamma : \mathit{fix}\ h \ \Downarrow\ (\Gamma, z \mapsto h\ z) : z$$

which is the same as first case we had, after removing the unnecessary binding for a in the first heap.

In other words, the two program states we obtain from the original expressions cannot be distinguished in any context. In the first case, the result will be delivered in variable z where z will contain the required cyclic structure in the heap. In the second case, the functional layer will expand *fix* h the first time the result is used. But the *LET* rule will create effectively the same structure in the heap in the very next step. In case the result is ignored, the first one will have a heap variable created which is never ever used. (It will simply be garbage collected.) In the second case, the heap will not have this variable at all. One can add a garbage collection rule to the functional layer to handle this issue more formally [5]. We refrain from doing so, in order to keep our rules as simple as possible.

6.3 Left shrinking

Left shrinking property states that:

$$\begin{aligned}&\mathit{fixIO}\ (\lambda x.\ q \gg\!= \ \lambda y.\ f\ x\ y)\\ &\qquad = \ q \gg\!= \ \lambda y.\ \mathit{fixIO}\ (\lambda x.\ f\ x\ y)\end{aligned}$$

where q is a free variable, bound to an arbitrary expression of the right type in the outer context. The crucial point is that x is not used by q. The types involved are: $q :: IO\ b$, and $f :: a \to b \to IO\ a$.

The very first steps in derivations of both handsides give:

$$\begin{aligned}&\Gamma : \mathit{fixIO}\ (\lambda x.\ q \gg\!= \ \lambda y.\ f\ x\ y)\\ \longrightarrow\ &(\text{FIXIO - FUN})\\ &(\Gamma,\ z \mapsto \bullet,\ a \mapsto z) : q \gg\!= \ \lambda y.\ f\ a\ y \gg\!= \ \mathit{update}_z\end{aligned}$$

and

$$\Gamma : q \gg\!= \ \lambda y.\ \mathit{fixIO}\ (\lambda x.\ f\ x\ y)$$

Now, if q is a divergent term, both derivations will diverge in the exact same way, that is both hand sides are equivalent. If q is normal, then by lemma 1, it will have a derivation of the form:

$$\Gamma : q \ \xrightarrow{\alpha}\ \Delta : \nu\vec{r}.(\mathit{return}\ \ qv\ \ |\ C)$$

where α is obtained by concatenating all the labels in the derivation, possibly empty. The C on the right hand side captures the passive containers that might be introduced in the derivation for q, along with the associated restrictions $\nu\vec{r}$. Since these containers will get copied in both sides in exactly the same way, we do not show them explicitly in what follows. Using the *HEAPEXT* and *EXTRUDE* rules silently, we can continue our derivations. On the lhs we get:

$$\begin{aligned}&(\Gamma,\ z \mapsto \bullet,\ a \mapsto z) : q \gg\!= \ \lambda y.\ f\ a\ y \gg\!= \ \mathit{update}_z\\ \xrightarrow{\alpha}\ &(\text{ASSUMPTION})\\ &(\Delta,\ z \mapsto \bullet,\ a \mapsto z) :\\ &\qquad \mathit{return}\ qv \gg\!= \ \lambda y.\ f\ a\ y \gg\!= \ \mathit{update}_z\\ \longrightarrow\ &(\text{LUNIT, FUN})\\ &(\Delta,\ z \mapsto \bullet,\ a \mapsto z,\ b \mapsto qv) : f\ a\ b \gg\!= \ \mathit{update}_z\end{aligned}$$

Let's look at the rhs:

$$\begin{aligned}&\Gamma : q \gg\!= \ \lambda y.\ \mathit{fixIO}\ (\lambda x.\ f\ x\ y)\\ \xrightarrow{\alpha}\ &(\text{ASSUMPTION - LUNIT})\\ &(\Delta,\ b \mapsto qv) : \mathit{fixIO}\ (\lambda x.\ f\ x\ b)\\ \longrightarrow\ &(\text{FIXIO - FUN})\\ &(\Delta,\ b \mapsto qv,\ z \mapsto \bullet,\ a \mapsto z) : f\ a\ b \gg\!= \ \mathit{update}_z\end{aligned}$$

Hence, the property holds *fixIO*. By this property, we are ensured that the recursive do-notation in Haskell will behave exactly like the usual do-notation, a key property in monadic value recursion [2].

6.4 Bekić property

Bekić property states that simultaneous recursion over multiple variables can be replaced by recursion on one variable at a time. For monadic value recursion, it states[3]:

$$\begin{aligned}&\mathit{fixIO}\ (\lambda t.\ \mathit{fixIO}\ (\lambda u.\ f\ (\mathit{fst}\ \ t,\ \mathit{snd}\ \ u)))\\ &\qquad = \mathit{fixIO}\ (\lambda v.\ f\ (\mathit{fst}\ \ v,\ \mathit{snd}\ \ v))\end{aligned}$$

where f is of type $(a,b) \to IO\ (a,b)$. On the left hand side, we have:

$$\begin{aligned}&\Gamma : \mathit{fixIO}\ (\lambda t.\ \mathit{fixIO}\ (\lambda u.\ f\ (\mathit{fst}\ \ t,\ \mathit{snd}\ \ u)))\\ \longrightarrow\ &(\text{FIXIO})\\ &(\Gamma,\ z \mapsto \bullet,\ a \mapsto z) :\\ &\qquad \mathit{fixIO}\ (\lambda u.\ f\ (\mathit{fst}\ \ a,\ \mathit{snd}\ \ u)) \gg\!= \ \mathit{update}_z\\ \longrightarrow\ &(\text{FIXIO})\\ &(\Gamma,\ z \mapsto \bullet,\ a \mapsto z,\ y \mapsto \bullet,\ b \mapsto y) :\\ &\qquad f\ (\mathit{fst}\ \ a,\ \mathit{snd}\ \ b) \gg\!= \ \mathit{update}_y \gg\!= \ \mathit{update}_z\end{aligned}$$

On the right hand side, we have:

$$\begin{aligned}&\Gamma : \mathit{fixIO}\ (\lambda v.\ f\ (\mathit{fst}\ \ v,\ \mathit{snd}\ \ v))\\ \longrightarrow\ &(\text{FIXIO})\\ &(\Gamma,\ z \mapsto \bullet,\ a \mapsto z) : f\ (\mathit{fst}\ \ a,\ \mathit{snd}\ \ a) \gg\!= \ \mathit{update}_z\end{aligned}$$

In both cases, f is given a tuple where both elements are bound to $\bullet$. If this term is divergent, both hand sides will diverge in the same way, as their arguments are indistinguishable as values. Similarly, if the call to f terminates with a value, the derivation will have a transition to *return fv*, for some term fv, and a new heap Δ. We will assume α represents the concatenated labels of all the transitions performed during this transition. Again, we do not show any passive containers that might be created along the way, they will be copied along in both cases in the exact same way.

Now, the lhs becomes:

$$\begin{aligned}\xrightarrow{\alpha}\ &(\text{ASSUMPTION})\\ &(\Delta,\ z \mapsto \bullet,\ a \mapsto z,\ y \mapsto \bullet,\ b \mapsto y) :\\ &\qquad \mathit{return}\ \mathit{fv} \gg\!= \ \mathit{update}_y \gg\!= \ \mathit{update}_z\\ \longrightarrow\ &(\text{LUNIT - UPDATE - LUNIT - UPDATE})\\ &(\Delta,\ z \mapsto y,\ a \mapsto z,\ y \mapsto \mathit{fv},\ b \mapsto y) : \mathit{return}\ z\end{aligned}$$

[3]Since Haskell types are lifted, we cannot simplify the right hand side to *fixIO f*. In Haskell, $(\perp, \perp)$ is not the same as $\perp$.

And the rhs becomes:
$\xrightarrow{\alpha}$ (ASSUMPTION)
$(\Delta,\ z \mapsto \bullet,\ a \mapsto z) : \mathit{return}\ \mathit{fv} \gg\!= \ \mathit{update}_z$
$\longrightarrow$ (LUNIT - UPDATE)
$(\Delta,\ z \mapsto \mathit{fv},\ a \mapsto z) : \mathit{return}\ z$

Which gives us the same term and an equivalent heap structure, just chase the pointer from z.

7 Conclusions and Future Work

We have given a semantics for the function *fixIO* of Haskell, and we have shown that it is is the fixed point operator for monadic-value recursion in the IO monad. Our approach presents a full operational semantics for a non-strict functional language extended with monadic IO primitives and references. Our contributions are:

- We show how a purely functional language and its semantics can be embedded into a language with monadic effects,
- We model sharing explicitly at all levels, giving an account of call by need in both the functional and the IO layers,
- We provide a semantics for *fixIO* and show that it indeed is a monadic value recursion operator.

Our work can be extended in several ways. The most obvious extension is the addition of threads and synchronized variables as in the awkward squad paper [11]. This extension does not present any challenges. (The addition of concurrency primitives will make the semantics non-deterministic. In that case, a derivation will be a set of transitions.) The difficulty, however, lies in extending the approach with asynchronous exceptions [8]. Although exceptions can be modeled nicely in the IO layer, we currently do not see a complementary way of capturing them in the functional layer using our method.

More work is needed in formalizing our arguments. Reasoning about the derivations in both IO and functional levels, as we did in the previous section, is a notoriously hard task. Although our system can easily be used to give semantics to concrete examples, it is harder to use it when reasoning about symbolic terms.

Study of other properties of *fixIO* is still pending. Although we have established the basic requirements, there are a number of other properties of interest as well (for instance the right shrinking law). Another open question is whether the semantics we have given for *fixIO* will satisfy the parametricity theorem for the type $(a \to IO\ a) \to IO\ a$. The parametricity theorem can further be used in establishing several derived properties for *fixIO*, such as swapping and pure right shrinking [3].

Detailed information about monadic value recursion can be found at `www.cse.ogi.edu/PacSoft/projects/rmb`.

8 Acknowledgements

We thank Simon Peyton Jones, Mark Shields and members of the OGI PacSoft Group for valuable discussions.

References

[1] Achten, P., and Peyton Jones, S. Porting the Clean Object I/O Library to Haskell. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages* (2000), pp. 194–213.

[2] Erkök, L., and Launchbury, J. A recursive `do` for Haskell: Design and implementation. Tech. Rep. CSE-00-014, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, August 2000.

[3] Erkök, L., and Launchbury, J. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00* (September 2000), ACM Press, pp. 174–185.

[4] Gordon, A. D. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. CUP, Sept. 1994.

[5] Launchbury, J. A natural semantics for lazy evaluation. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina* (1993), pp. 144–154.

[6] Launchbury, J., Lewis, J., and Cook, B. On embedding a microarchitectural design language within Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99)* (1999), pp. 60–69.

[7] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *Lisp and Symbolic Computation 8*, 4 (Dec. 1995), 293–341.

[8] Marlow, S., Peyton Jones, S. L., Moran, A., and Reppy, J. Asynchronous exceptions in haskell. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)* (Snowbird, Utah, June 20–22 2001).

[9] Milner, R. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, May 1999.

[10] Nordlander, J. *Reactive Objects and Functional Programming.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1999.

[11] Peyton Jones, S. L. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction* (2001), T. Hoare, M. Broy, and R. Steinbruggen, Eds., IOS Press, pp. 47–96.

[12] Peyton Jones, S. L., and Hughes, J. (Editors.) Report on the programming language Haskell 98, a non-strict purely-functional programming language. Available at: `http://www.haskell.org/onlinereport`, Feb. 1999.

[13] Peyton Jones, S. L., and Wadler, P. Imperative functional programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina* (1993), pp. 71–84.