Theoretical Informatics and Applications
Theoret. Informatics Appl. **36** (2002) 155-180
DOI: 10.1051/ita:2002008

SEMANTICS OF VALUE RECURSION FOR MONADIC INPUT/OUTPUT *

Levent Erkök[1], John Launchbury[1,2] and Andrew Moran[2]

Abstract. Monads have been employed in programming languages for modeling various language features, most importantly those that involve side effects. In particular, Haskell's IO monad provides access to I/O operations and mutable variables, without compromising referential transparency. Cyclic definitions that involve monadic computations give rise to the concept of *value-recursion*, where the fixed-point computation takes place only over the values, without repeating or losing effects. In this paper, we describe a semantics for a lazy language based on Haskell, supporting monadic I/O, mutable variables, usual recursive definitions, and value recursion. Our semantics is composed of two layers: A natural semantics for the functional layer, and a labeled transition semantics for the IO layer.

Mathematics Subject Classification. 68N18, 68Q55, 18C15.

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 [27,28]. Together with the IO monad, a rather mysterious function called *fixIO*, with type $\forall\tau.(\tau \to IO\ \tau) \to IO\ \tau$, was also introduced. As Achten and Peyton Jones point out, *fixIO "...allows us to manipulate results [of IO computations] that are not yet computed, but lazily available"* [1] (Sect. 4.1). In this regard, the functionality provided by *fixIO* is similar to that of *fixST* associated with the state monad [18]. Both *fixIO* and *fixST* provide fixed-point operators that enable recursion resulting from the values of monadic actions. Later work tried to explain the behavior of such fixed-point

* *This is a revised and extended version of a paper that appeared in FICS'01 [7].*

[1] OGI School of Science and Engineering, OHSU; e-mail: erkok@cse.ogi.edu

[2] Galois Connections, Inc.

© EDP Sciences 2002

operators from an axiomatic point of view [6]. It was noted that a generic fixed-point operator, one that would work regardless of the underlying effect, is not available. Instead, one has to specify a suitable fixed-point operator for each different kind of monadic effect. The function *fixIO* was conjectured to be the corresponding operator for Haskell's IO monad, pending a detailed treatment.

To explore this conjecture further, we need to understand the operation of *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 [26], in the spirit of monadic transition systems that were previously studied by Gordon [9]. 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 [22]. Peyton Jones's work used an embedding of a denotational semantics for the functional layer into the IO layer. However, it did not capture sharing implied by lazy evaluation, the precise interaction of IO and functional layers, and value recursion. Such an approach is fine, as long as one is interested in the big picture. If, on the other hand, one wants to capture value recursion *via fixIO*, it becomes necessary to be explicit about the details of the embedding. One aim of 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 [26]. The semantics for the functional layer is based on the natural semantics for lazy evaluation of Launchbury [16]. 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 remainder of this paper is structured as follows. After reviewing background material on value recursion, we motivate the use of *fixIO* with some simple examples. Then, we present a language with monadic I/O, mutable variables, and value recursion, together with a two layer semantics. We conclude with a discussion of the properties satisfied by *fixIO*, summary of our results, and pointers for future work.

1. Related Work and Background Material

Usual domain theoretic treatment of recursion relies on the least fixed-points of continuous functions. In the implementations of programming languages, recursion is generally achieved *via* cyclic structures [2, 10]. Traced monoidal categories, proposed by Joyal *et al.* [15], provide a framework for abstractly capturing such feedback operations. As shown in Hasegawa's thesis [11], every traced cartesian category admits a Conway fixed-point operator [32], and vice versa, satisfactorily explaining recursion resulting from cyclic sharing. A particularly nice summary of the categorical treatment of fixed-point operators can be found in Simpson and Plotkin's recent work [32].

On a parallel thread, Moggi introduced monads to computer science as a way of structuring denotational semantics [23]. Use of monads influenced functional programming community deeply, resulting in a surge of practical applications. Most

importantly, Haskell's I/O mechanism was redesigned to use monads [27]. One particular application domain, however, surfaced the need for a different notion of recursion in monadic computations: Hardware design languages, such as Hawk and Lava, employ monads to model circuit elements [4,21]. An important shortcoming of the monadic approach is the difficulty in modeling feedback loops in circuits, a fundamental circuit design principle. Launchbury *et al.* [17] noticed that a new kind of fixed-point operator is needed, one that would neither repeat nor lose the effects, but perform recursion only over the values. Later work coined the name *value recursion* for this concept [5,6].

What is the right framework for modeling value recursion? Recalling the correspondence between usual fixed-point operators and traced monoidal categories, it is natural to ask whether we can build a corresponding framework for value recursion as well. The answer to this question is still open; we only review the development so far. One way of modeling value recursion for a strong monad over a cartesian category is to consider a trace operator on the corresponding Kleisli category. Unfortunately, the monoidal requirement is too strong: unless the underlying monad is commutative, the Kleisli category will only be premonoidal [30], and hence the notion of trace is simply not applicable. Hasegawa noted that the extension of traces to premonoidal categories might prove interesting [11] (Chap. 9). Jeffrey followed upon this idea, but he worked with a notion of partial traces, limiting recursion only to a certain class of maps [13].

More recently, Paterson [25] considered value recursion for *arrows*, a generalization of monads [12]. The adaptation of trace axioms, however, turned out to be too strong for various arrows. More specifically, certain arrows failed to satisfy the right tightening law required by trace axioms. Indeed, right tightening law has been observed to be too strong in our earlier work as well (where it was called right-shrinking) [6]. Independently of Paterson, Benton and Hyland tried to generalize traces to premonoidal categories as well [3]. They also attempted to generalize the notion of fixed-point operators to Freyd categories, trying to establish a correspondence with the traced premonoidal case. However, their axiomatization fails to provide value recursion operators for several monads of practical interest, most notably exceptions, lazy lists, strict state, and the IO monad, the subject matter of the present paper. In all these cases, the right tightening axiom fails.

Another common theme in both Paterson's and Benton and Hyland's axiomatization is the weakening of the sliding axiom. (Recall that sliding axiom for traces implies swapping the order of computations, which is simply not acceptable in the premonoidal case [15], as order does matter in performing effects.) Hence, both Paterson and Benton and Hyland weakened the sliding axiom so that one of the involved arrows is central, *i.e.*, one that would commute with any other arrow in the category. However, even this weakened version of sliding is too strong for value recursion. Similar to the right-tightening axiom, the premonoidal version of sliding is unsatisfiable for many practical monads, including exceptions, lazy lists, strict state, and the IO monad [5].

For the purposes of the current paper, we will use our earlier equational presentation of properties of value recursion operators, interpreted in the usual domain

theoretic model. This is a natural path to follow, given that our aim is to reason about real Haskell programs. When interpreted with respect to the IO monad and the function *fixIO*, these laws are [5]:

1. *Strictness:* let $f :: \tau \to IO\ \tau$ be a strict function. Then, *fixIO* f must be $\bot$. Strictness is an obvious extension of the corresponding property of usual fixed-points, *i.e.*, it ensures that the fixed-point computation does not introduce spurious values or effects. Symbolically:

$$f\ \bot_\tau = \bot_{IO\,\tau} \quad \longrightarrow \quad \mathit{fixIO}\ f\ = \bot_{IO\,\tau}. \tag{1}$$

2. *Purity:* consider the composition $\mathit{return}\ \cdot\ h :: \tau \to IO\ \tau$, where h is a function of type $\tau \to \tau$. That is, functions with no effects. In this case, the value-recursion computation should simply lift the pure fixed-point value to the monadic world. Symbolically[3]:

$$\mathit{fixIO}\ (\mathit{return}\ \cdot\ h)\ = \mathit{return}\ (\mathit{fix}\ \cdot\ h). \tag{2}$$

3. *Left shrinking:* consider a recursive computation that can be decomposed into two parts, where the initial segment does not make use of the fixed-point value. In this case we can shrink the recursive loop from the left, *i.e.*, pull the initial segment out of the recursive loop. Symbolically:

$$\mathit{fixIO}\ (\lambda x.\ q \gg\!= \lambda y.\ f\ x\ y)\ = q \gg\!= \lambda y.\ \mathit{fixIO}\ (\lambda x.\ f\ x\ y) \tag{3}$$

   where x does not appear free in q. The types involved are: $q :: IO\ \sigma$, and $f :: \tau \to \sigma \to IO\ \tau$.

For a graphical presentation of these laws (and others), the reader is referred to our earlier work, and Paterson's version for arrows [5, 6, 25]. In Section 6, we will review each one of these laws to establish that *fixIO* is indeed the required value recursion operator for the IO monad with respect to our semantics.

2. Motivating Examples

We start by considering several Haskell program fragments, demonstrating the use of the function *fixIO*. We will allow ourselves to use Haskell's *do*-notation for notational convenience [27].

[3] We use the usual Haskell notation: $\mathit{return} :: \tau \to IO\ \tau$ is the unit of the monad, while the infix operator $\gg\!= :: IO\ \tau \to (\tau \to IO\ \sigma) \to IO\ \sigma$ serves as the bind [27].

Example 2.1. Our first example shows the interaction of *fixIO* with input operations:

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

When we run this computation, a character will be read from the standard input, say `a`. Then, the computation will immediately deliver an infinite list of `a`'s[4]. 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: (i) the action *getChar* is executed only once, and (ii) the computation terminates immediately 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 provides a means for recursively defining values resulting from monadic actions. That is, it allows naming results of computations that will only be available later on, similar to the usual fixed-point operator. For instance, in the expression given above, 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 semantics of the pure expression:

$$\mathbf{let}\ cs = \text{'a'} : cs\ \mathbf{in}\ cs$$

which is a more convenient way of writing: $\mathit{fix}\ (\lambda cs.\ \text{'a'} : cs)$, where *fix* is the usual fixed-point operator. Except, of course, in the *fixIO* case the character in the list is determined by the call to *getChar*, *i.e.*, it depends on the actual input available when we run the computation.

Example 2.2. Here is a Haskell expression showing the interaction of *fixIO* with mutable variables:

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

In this expression, we allocate a cell in which we store the value of the variable x, before we know what that value really is. The value of x, determined through the fixed-point computation, is 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 and get back the lazily computed list of 1's. Although this example might look superficial, it basically captures the essence of cyclic structures with mutable nodes. For instance, we have previously shown how such a technique can be used to implement doubly-linked-lists, where each node holds a mutable boolean value [6]. As shown by Nordlander, a similar situation arises in object oriented programming, when several objects need to refer to each other cyclically [24].

[4]Note that, by applying the left shrinking and purity laws of the previous section, we can reduce this expression to $\mathit{getChar} \gg\!= \lambda c.\ \mathit{return}\ (\mathit{fix}\ (\lambda cs.\ c{:}cs))$, guaranteeing the described behavior axiomatically. Of course, we have not yet established that these two properties hold for *fixIO*, but we will do so in Section 6.

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

3. The Language

In this section, we define a language based on Haskell [27], supporting monadic IO primitives, mutable variables, usual recursive definitions, and value recursion.

Notation 3.1. We use the following naming conventions for variables:

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

To simplify the discussion, we syntactically distinguish between heap and mutable variables: They are drawn from different alphabets.

Definition 3.2. (*Terms and values*) Terms and values are defined mutually recursively by the following grammars:

$$\begin{array}{lrcl} (\textit{Terms}) & M,\ N & ::= & x \\ & & | & V \\ & & | & M\ N \\ & & | & \textbf{let}\ \vec{x} = \vec{M}\ \textbf{in}\ N \\ & & | & \textbf{case}\ M\ \textbf{of}\ \{c_i\ \vec{x}_i \to N_i\} \\ (\textit{Values}) & V & ::= & c\ x_1\ x_2\ \ldots\ x_i \\ & & | & \lambda x.\ M \\ & & | & \textit{return}\ M \mid M \gg\!= N \\ & & | & \textit{getChar} \mid \textit{putChar}\ M \\ & & | & \textit{fixIO}\ M \mid \textit{update}_z\ M \\ & & | & r \\ & & | & \textit{newIORef}\ M \\ & & | & \textit{readIORef}\ M \\ & & | & \textit{writeIORef}\ M\ N. \end{array}$$

The function $\textit{update}_z$, associated with the heap variable z, cannot appear in a valid input program, and it is never the result of any program either. It is only used internally, in giving a semantics to *fixIO*. We will explain its role in detail later. All other constructs have the same meaning and type as they do in Haskell. Note that IO actions are values as far as the purely functional world is concerned.

For the purposes of this paper, we only work with well-typed terms, and ignore the issues of type checking and inference. We assume that the usual Haskell rules apply to determine well typed terms. (Typing of Haskell programs has been discussed in detail in the literature [14,27].) Notice that *return*, $\gg\!=$, *fixIO*, etc., are polymorphic constants. As usual, **let** expressions provide recursive (and possibly polymorphic) bindings.

A constructor c of arity i is treated as a function $\lambda x_1 \ldots x_i.\ c\ x_1 \ldots x_i$, which becomes a value of its own when fully applied. This case is captured by the first

alternative in the definition of values, where c is assumed to have arity i. We model constants as nullary constructors, that is, numbers, characters, etc., are treated as constructors with zero arity. (As a notational hint, we will use the letter k to refer to constants.)

Remark 3.3. It is worth noting that the grammar we gave describes the syntax for the reduced terms of our language rather than the concrete syntax that we will allow ourselves to use. In particular, we will freely use the do-notation of Haskell and pattern bindings in λ-abstractions [27]. In each case, however, the translation to the core language will be trivial.

Definition 3.4. (*IO and pure terms*) A well-typed term of type $IO\ \tau$, for some type τ, is called an IO term. All other terms are called pure.

Definition 3.5. (*Terminal values*) A value is called terminal if it has one of the following forms:

- $c\ x_1\ x_2 \ldots x_i$, where c is a constructor of arity i;
- $\lambda x.M$;
- $return\ M$;

where M is an arbitrary term in the second and third cases.

Definition 3.6. (*Heaps*) A heap is a finite partial function from heap variables to terms extended with a special black hole value $\bullet$:

$$\Gamma\ ::\ \mathit{Heap\ Variables} \rightharpoonup \mathit{Terms} \cup \{\bullet\}.$$

A heap binding can be polymorphically typed. A black hole binding, such as $z \mapsto \bullet$, indicates that the variable is known but not directly accessible. Intuitively, $\bullet$ is a detectable bottom.

Notation 3.7. Although heaps are functions, we will allow ourselves to use the set notation freely on them: the notation $x \mapsto M \in \Gamma$ simply states that Γ maps x to M. The empty heap is denoted $\{\}$. The notation $(\Gamma, x \mapsto M)$ denotes the heap Γ extended with a new binding $x \mapsto M$. In this case, x cannot be already bound in Γ, but might appear free in M.

Since our language allows input operations, the meaning of a term might depend on the input stream it receives while being run. To accommodate this view, we have to consider terms and input streams together.

Definition 3.8. (*Input streams*) An input stream is a list of characters, not necessarily finite.

Notation 3.9. We will use the Haskell list notation to denote input streams. [] (or `""`) denotes the empty input stream, *i.e.*, the case when the input is exhausted. Otherwise, a stream is of the form $(c : I)$, where c is a character and I is an input stream.

Definition 3.10. (*Term and program states*) A running program is identified by its program state, which consists of an input stream, a heap and a term state:

(*Terms States*)	P	::=	M	Current term
		$\mid$	$P \mid \langle x \rangle_r$	Passive container
		$\mid$	$\nu r.P$	Restriction.

We use the notation $I : \Gamma : P$ to denote program states.

A term state is simply the current term under consideration, together with a number of passive containers. A passive container $\langle x \rangle_r$ represents a mutable variable named r, which holds a heap variable x. (We only store heap variables in these containers; the actual contents are stored in the heap.) Restrictions convey the scoping information for mutable variables. Notice that a program state contains enough information to capture a program in execution.

Remark 3.11. To reduce clutter, we will generally skip the bits of the program state that are not needed in the discussion, especially when we write our rules. That is, we will use $\Gamma : P$, if the input stream is irrelevant, and similarly $I : P$, when the heap is not needed. There is no chance of confusion, however, because we only use capital Greek letters for heaps and never skip the term state.

Definition 3.12. (*The functions bn and fn*) 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 function *fn* is defined for term states and heaps. Given a term state, *fn* returns the set of free variables in it. A heap variable x is free if it is not in the scope of a λx binding. A mutable variable r is free if it is not in the scope of a νr binding. For a heap Γ, $fn(\Gamma) = \bigcup\{fn(M) \mid x \mapsto M \in \Gamma\} - bn(\Gamma)$. We treat *fn* as a variable-arity function: $fn(A, B)$ means $fn(A) \cup fn(B)$, and similarly for more arguments.

Definition 3.13. (*Slice of a heap*) The slice of a heap Γ, with respect to a term state P, written Γ/P, is the subset of Γ that is reachable from the free names of P. More precisely, for a given Γ and P, let

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

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

$$\Gamma/P = \{x \mapsto M \mid x \in S \wedge x \mapsto M \in \Gamma\}. \tag{4}$$

Definition 3.14. (*Closed program states*) A program state $S : \Gamma : P$ is closed if $fn(\Gamma) = \emptyset$, and $fn(P) \subseteq bn(\Gamma)$. (Note that if the second condition is satisfied, no mutable variable in P can be free.)

Definition 3.15. (*Type of a program state*) Let $S : \Gamma : P$ be a closed program state, and let M be the term associated with P. We say that $S : \Gamma : P$ has type τ, and write $(S : \Gamma : P) :: \tau$, when M has type τ when typed in the heap Γ.

Definition 3.16. (*Terminal program state*) A program state $S : \Gamma : P$ is terminal if the term associated with P is terminal (Def. 3.5).

4. Semantics

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

Given a term, we need to be able to extract the part that is going to be executed next. We use contexts to guide this search:

Definition 4.1. (*Execution Contexts*) Execution contexts are described by the following grammar:

$$\begin{array}{lrcl} (\textit{Execution Contexts}) & E & ::= & [\cdot] \\ & & | & E \gg\!= M. \end{array}$$

An execution context is a term with one hole, where the hole itself is filled with a term. The notation $E[M]$ denotes the context E filled with the term M. An empty context is one where there are no $\gg\!=$'s, as captured by the first alternative. Otherwise, the context is non-empty, *i.e.*, it is some IO action followed by others[5]. If the context is empty, the term filling the context might be pure.

4.1. IO layer

Figure 1 gives the transition rules for the IO layer.

A rule is a (possibly labeled) transition from a program state to another. The label "!c" indicates that the character c is printed on standard output, and the one labeled "?c" indicates that the next character from the input stream (which happens to be c) is consumed.

To simplify the notation, we use a couple of conventions in writing our rules (which are going to be formalized in Sect. 4.4). Rather than a verbal explanation, we will consider several illustrative examples:

Example 4.2. Consider the program state

$$\texttt{"ab"} : \Gamma : \mathit{getChar} \gg\!= \mathit{putChar}$$

for some heap Γ. The term state consists of the single term $\mathit{getChar} \gg\!= \mathit{putChar}$. When we match this term to the context grammar given in Definition 4.1, we see that there are two possibilities. Either we can have the empty context, filled with the term $\mathit{getChar} \gg\!= \mathit{putChar}$, or the context $[\cdot] \gg\!= \mathit{putChar}$, filled with the term *getChar*. Upon inspection of our rules, we see that only the second has a chance of matching a rule, namely *GETC*. Since the *GETC* rule requires the input stream

[5]Other authors use the term *evaluation context* for this concept [8]. We prefer the term execution, since a non-empty context can only be filled by an IO action which is going to be executed next.

$$E[\mathit{putChar}\ c] \xrightarrow{!c} E[\mathit{return}\ ()] \quad (\mathit{PUTC})$$
$$(c : I) : E[\mathit{getChar}] \xrightarrow{?c} I : E[\mathit{return}\ c] \quad (\mathit{GETC})$$

$$E[\mathit{return}\ N \gg\!= M] \longrightarrow E[M\ N] \quad (\mathit{LUNIT})$$

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

$$E[\mathit{readIORef}\ r] \mid \langle x \rangle_r \longrightarrow E[\mathit{return}\ x] \mid \langle x \rangle_r \quad (\mathit{READIO})$$

$$\frac{y \notin \mathit{bn}(\Gamma)}{\Gamma : E[\mathit{writeIORef}\ r\ N] \mid \langle x \rangle_r \longrightarrow (\Gamma, y \mapsto N) : E[\mathit{return}\ ()] \mid \langle y \rangle_r} \quad (\mathit{WRITEIO})$$

$$\frac{z \notin \mathit{bn}(\Gamma)}{\Gamma : E[\mathit{fixIO}\ M] \longrightarrow (\Gamma, z \mapsto \bullet) : E[M\ z \gg\!= \mathit{update}_z]} \quad (\mathit{FIXIO})$$

$$(\Gamma, z \mapsto \bullet) : E[\mathit{update}_z\ M] \longrightarrow (\Gamma, z \mapsto M) : E[\mathit{return}\ z] \quad (\mathit{UPDATE})$$

FIGURE 1. Semantics: IO layer.

to be of the form $(c : I)$, we have to make sure that we have a non-empty stream. Because `"ab"` is not empty, the *GETC* rule is applicable. Hence, we end up with the transition:

$$\texttt{"ab"} : \Gamma : \mathit{getChar} \gg\!= \mathit{putChar} \xrightarrow{?\texttt{a}} \texttt{"b"} : \Gamma : \mathit{return}\ \texttt{'a'} \gg\!= \mathit{putChar}.$$

Note that the *GETC* rule does not make use of the heap, hence it is not even mentioned. The heap is simply carried across unchanged.

Example 4.3. Consider what happens when we continue the preceding example. Again, there are two possible choices for the context. The empty context, filled with the term *return* `'a'` $\gg\!=$ *putChar*, or the context $[.] \gg\!=$ *putChar*, filled with the term *return* `'a'`. Unlike the preceding case, however, the first choice matches the *LUNIT* rule, while the second one does not match any. Since the *LUNIT* rule does not constrain the input stream or the heap in any way, it is applicable. Hence, we end up with the transition:

$$\texttt{"b"} : \Gamma : \mathit{return}\ \texttt{'a'} \gg\!= \mathit{putChar} \longrightarrow \texttt{"b"} : \Gamma : \mathit{putChar}\ \texttt{'a'}.$$

Since *PUTC* rule does not make use of the input stream or the heap, it does not explicitly mention them. They are both simply copied. It should now be obvious that the next transition is:

$$\texttt{"b"} : \Gamma : \mathit{putChar}\ \texttt{'a'} \xrightarrow{\texttt{!a}} \texttt{"b"} : \Gamma : \mathit{return}\ ()$$

and there are no more transitions from this state, as none of the rules match.

Example 4.4. Consider the program state $I : \Gamma : \mathit{newIORef}\ 5 \gg\!= \mathit{readIORef}$, for some I and Γ. The only matching choice for the context is $[\cdot] \gg\!= \mathit{readIORef}$, with the term *newIORef 5* filling the hole. The *NEWIO* rule applies. To satisfy the precondition of this rule, we have to pick variables r and x such that $r \notin \mathit{fn}(\mathit{newIORef}\ 5 \gg\!= \mathit{readIORef})$ and $x \notin \mathit{bn}(\Gamma)$. We simply pick fresh variables to satisfy these requests. Let us call them r and x for simplicity. We end up with the transition:

$$\begin{aligned} I : \Gamma : \mathit{newIORef}\ 5 \gg\!= \mathit{readIORef} \\ \longrightarrow \quad I : (\Gamma,\ x \mapsto 5) : \nu r.(\mathit{return}\ r \gg\!= \mathit{readIORef}\ \mid \langle x \rangle_r). \end{aligned}$$

Example 4.5. We will continue with the previous example. Clearly, we want to apply the *LUNIT* rule, but it is not clear how we get over the restriction νr. If we look at the *LUNIT* rule, we see that only a term in context is specified (as in all rules except *READIO* and *WRITEIO*). The convention we adopt in this case is the following: if a rule only mentions a term in a context in the term state position, then we consider the term associated with the current program state and try to match it. Any remaining restrictions, passive containers, etc. are copied along. That is, the application of the *LUNIT* rule in this case yields:

$$\begin{aligned} I : (\Gamma,\ x \mapsto 5) : \nu r.(\mathit{return}\ r \gg\!= \mathit{readIORef}\ \mid \langle x \rangle_r) \\ \longrightarrow \quad I : (\Gamma,\ x \mapsto 5) : \nu r.(\mathit{readIORef}\ r\ \mid \langle x \rangle_r). \end{aligned}$$

Example 4.6. Finally we show how to handle rules that have both a term in context and a passive reference mentioned in their left hand sides, namely the *WRITEIO* and *READIO* rules. Continuing the previous example, we see that the *READIO* rule needs to be applied, which requires a term of the form *readIORef r* next to a passive container named r. In this case, our convention is the following: if a rule mentions a term in context next to a passive container, then a program state matches it if and only if we can show that the term associated with it matches the term in context, and we are next to the corresponding passive container. In our case, we get the following transition:

$$\begin{aligned} I : (\Gamma,\ x \mapsto 5) : \nu r.(\mathit{readIORef}\ r\ \mid \langle x \rangle_r) \\ \longrightarrow \quad I : (\Gamma,\ x \mapsto 5) : \nu r.(\mathit{return}\ 5\ \mid \langle x \rangle_r). \end{aligned}$$

Remark 4.7. The careful reader must have noticed that it is not necessarily the case that we will always have the required passive container positioned nicely. For

example, if we start with the program state

$$[\,] : \{\} : \mathit{newIORef}\ \ 0 \gg\!= \ \ \lambda r.\ \mathit{newIORef}\ \ 1 \gg\!= \ \ \lambda s.\ \mathit{readIORef}\ \ r$$

we will end up with:

$$[\,] : \{x \mapsto 0,\ y \mapsto 1\} : \nu r.(\nu s.(\mathit{readIORef}\ \ r\ \mid \langle y \rangle_s) \mid \langle x \rangle_r).$$

Clearly, we want to apply the *READIOREF* rule here as well. Alas, the rule does not match. In these cases, we will need to use structural rules, which provide means for transforming the program state into an equivalent one such that there is an applicable rule. Structural rules are covered in Section 4.4.

Some comments about the *FIXIO* rule are in order. The function *fixIO* 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 black hole. Otherwise, z could be passed around, stored in data structures, etc.: note that it is just a normal heap variable. Once the function call completes, we update the heap variable z by the result, effectively tying the knot by an application of the *UPDATE* rule. In summary, z holds the value of the entire computation, which might in turn depend lazily on its own value, *i.e.*, it is recursively defined.

Although the rules of our IO layer are quite similar to those given by Peyton Jones [26], the following differences are worth mentioning:

- we keep track of the input stream explicitly, rather than assuming that standard input will be consulted whenever a *getChar* is executed;
- as in the natural semantics of Launchbury [16], we keep track of a separate global heap to store values of variables;
- unlike Peyton Jones's semantics, our reference cells only store heap variables, rather than arbitrary terms. This restriction is necessary in order to model sharing implied by lazy evaluation.

4.2. Functional layer

Our rules for the functional layer, given in Figure 2, follow Launchbury's natural semantics for lazy evaluation closely [16]. Note that none of the rules in this layer mention the input stream, as it is irrelevant at this layer. Also, we use the notation $\Downarrow$, rather than $\longrightarrow$, for reductions. Compared to the IO layer, where we have a small steps semantics, the rules in the functional layer encode a big step natural semantics.

Compared to Launchbury's natural semantics [16], some minor differences worth mentioning are:

- we introduce a new black hole binding;

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

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

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

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

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

FIGURE 2. Semantics: functional layer.

- 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 *APP* rule, we require $w \notin bn(\Gamma)$. In the *LET* rule, we rename all bound variables $x_1 \ldots x_n$ to $\hat{x}_1 \ldots \hat{x}_n$ so that there will not be any name clashes in the heap when we do the additions. Similarly, the term $\hat{M}_i$ denotes the term M_i, where each occurrence of x_i is replaced by $\hat{x}_i$. (Similarly for $\hat{N}$.) The *VAR* rule is not applicable if the variable being looked up is bound to $\bullet$ in the heap. If this case ever occurs, the derivation will simply terminate with failure, corresponding to a detectable black hole.

We refrain from going into details of this layer, as such systems are rather well studied in the literature. The interested reader is referred to Launchbury's original exposition [16], and Sestoft's work on abstract machines based on such systems [31].

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. (Note that the *PUTC* rule of the IO-layer only

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

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

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

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

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

applies when the argument to *putChar* is a constant.) 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 allows handling of applications, **let** and **case** expressions, and variable lookups. This interaction is provided by embedding the functional world into the IO world, as modeled by the *FUN* rule. In all these rules, M is assumed to be a non-value: the functional layer is consulted to reduce M to a value.

4.4. Structural rules

Finally, we need a set of structural rules to shape our derivations. As discussed in Remark 4.7, structural rules do not perform evaluation steps as do the other rules, but they might be necessary in order to transform a program state to an equivalent one such that one of the transition rules can apply.

The first set of structural rules, presented in Figure 4, state that certain program states are equivalent to others. As usual, we mention input streams and heaps only when they are relevant. The *ALPHA* rules state that heap and mutable variables can be renamed at will, *i.e.*, we do not distinguish program states that differ only in the names of variables. (Substitution on heaps is defined as $\Gamma[x/y] = \{z \mapsto M[x/y] \mid z \mapsto M \in \Gamma\}$.) Note that we do not need a side condition of the form $s \notin bn(\Gamma)$ in *ALPHA1*, since only heap variables can be bound in the heap.

The *HEAPEXT* rule states that we can add new bindings, as long as they do not interfere with existing bindings. See Section 6 for an example use of this

$$\frac{s \notin fn(P)}{\Gamma : \nu r.P \;\equiv\; \Gamma[s/r] : \nu s.P[s/r]} \quad (ALPHA1)$$

$$\frac{y \notin fn(\Gamma, M, P) \wedge y \notin bn(\Gamma)}{(\Gamma, x \mapsto M) : P \;\equiv\; (\Gamma, y \mapsto M)[x/y] : P[x/y]} \quad (ALPHA2)$$

$$\frac{x \notin bn(\Gamma) \;\wedge\; x \notin fn(\Gamma,\, P)}{\Gamma : P \equiv (\Gamma,\; x \mapsto M) : P} \quad (HEAPEXT)$$

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

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

FIGURE 4. Semantics: structural rules, Part I.

rule[6]. The rules *COMM*, *ASSOC* and *SWAP* state obvious equivalences. Finally *EXTRUDE* shows how we can manipulate the scoping of reference variables. The side condition in the *EXTRUDE* rule guarantees that no dangling references will be created. (See Ex. 5.3 for details.)

The second set of structural rules, presented in Figure 5, formalize our conventions in applying the rules. The first four rules simply state that we can concentrate on the relevant bits of the derivation and add the extra bits later on. And finally, *EQUIV* states that we only need to consider program states up to equivalence when performing transitions.

Example 4.8. We will reconsider the example discussed in Remark 4.7. Recall that we had the program state:

$$[\,] : \{x \mapsto 0,\; y \mapsto 1\} : \nu r.(\nu s.(\mathit{readIORef}\;\; r \;\mid \langle y \rangle_s) \mid \langle x \rangle_r).$$

[6]We can also add a garbage collection rule to get rid of unreachable heap variables and passive containers. We will avoid such a rule for the sake of brevity, as it is not essential for our current purposes.

$$\frac{P \xrightarrow{\alpha} Q}{\Gamma : P \xrightarrow{\alpha} \Gamma : Q}\ (HEAPIN) \qquad \frac{P \xrightarrow{\alpha} Q}{I : P \xrightarrow{\alpha} I : Q}\ (STREAMIN)$$

$$\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R}\ (PAR) \qquad \frac{P \xrightarrow{\alpha} Q}{\nu r.P \xrightarrow{\alpha} \nu r.Q}\ (NU)$$

$$\frac{\Gamma : P \equiv \Delta : P' \qquad \Delta : P' \xrightarrow{\alpha} \Theta : Q' \qquad \Theta : Q' \equiv \Sigma : Q}{\Gamma : P \xrightarrow{\alpha} \Sigma : Q}\ (EQUIV)$$

FIGURE 5. Semantics: structural rules, Part II. The label α ranges over empty transitions as well.

By applying *EXTRUDE, ASSOC, COMM, ASSOC* and *READIOREF* rules (and by appropriate applications of the rules in Figure 5 to enable them), we get:

$$\begin{array}{rl} \equiv & [\,] : \{x \mapsto 0,\ y \mapsto 1\} : \nu r.(\nu s.((\mathit{readIORef}\ \ r\ \mid \langle y \rangle_s) \mid \langle x \rangle_r)) \\ \equiv & [\,] : \{x \mapsto 0,\ y \mapsto 1\} : \nu r.(\nu s.((\mathit{readIORef}\ \ r\ \mid \langle x \rangle_r) \mid \langle y \rangle_s)) \\ \longrightarrow & [\,] : \{x \mapsto 0,\ y \mapsto 1\} : \nu r.(\nu s.((\mathit{return}\ \ x\ \mid \langle x \rangle_r) \mid \langle y \rangle_s)). \end{array}$$

There are no matching rules for the resulting program state. We can apply structural rules again, but none will give us a program state where a non-structural rule can apply.

Remark 4.9. One can extend $\equiv$ to an equivalence relation on program states by simply adding rules to make it reflexive and transitive. However, the current definition of $\equiv$ given in Figure 4 is simply too crude to be useful for this purpose. Intuitively, we want to be able to identify program states if their "observable behavior" are the same [9,20,29]. We defer a formal development of this possibility to future work.

4.5. Meaning of program states

The meaning of a closed program state is its derivation:

Definition 4.10. (*Derivations*) Let $I : \Gamma : P$ be a closed program state. The derivation for $I : \Gamma : P$ is a sequence of labeled transitions, where at each step a rule is applied. Structural rules can be applied at any time, as long as they trigger the application of a non-structural rule. The derivation continues until there are no applicable rules.

Simple inspection of our rules reveals that we have a deterministic system modulo the structural rules. That is, given a program state there is at most one non-structural rule that can apply to it.

Definition 4.11. (*Effect of a derivation*) The effect of a derivation is the concatenation of its transition labels. Empty transitions do not contribute to the effect.

The effect of a program state is simply a (possibly infinite) list, where each element is of the form "?c" or "!c" for some character c.

Notation 4.12. As usual, $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$. We will shorten multiple steps of derivations using the notation $I : \Gamma : P \xrightarrow{\alpha}{}^* I' : \Gamma' : P'$.

Definition 4.13. (*Divergent and normal program states*) A closed program state $I : \Gamma : P$ is called divergent if the derivation starting from $I : \Gamma : P$ either

- continues indefinitely (*i.e.*, we never run out of non-structural rules to apply);
- or, gets stuck in a non-terminal program state (Definition 3.16) where no non-structural rule applies.

Otherwise, $I : \Gamma : P$ is called normal.

Example 4.14. It is easy to come up with divergent terms. For instance, one can show that the derivation for:

$$I : \Gamma : \mathbf{let}\ \mathit{loop} = \mathit{putChar}\ \texttt{'a'} \gg \mathit{loop}\ \mathbf{in}\ \mathit{loop} \tag{5}$$

diverges, since we never run out of rules to apply. However, the derivation for:

$$I : \Gamma : \mathbf{let}\ x = x\ \mathbf{in}\ x \tag{6}$$

will diverge by getting stuck. The *FUN* rule will never fire, because there are no reductions for this term in the functional layer. (Notice that the first application of the *VAR* rule will result in $I : (\Gamma,\ x \mapsto \bullet) : x$, but no other rule will apply since the *VAR* rule is only applicable when the binding is not a black hole.) Similarly, a derivation can get stuck *via* the use of the *FIXIO* rule (which introduces a black hole binding in the heap). A final possibility is the application of the *GETC* rule when the input stream is empty.

Lemma 4.15. *(Derivations for normal program states) Let $I : \Gamma : P$ be a normal program state. The derivation starting at this state will take the form*

$$I : \Gamma : P \xrightarrow{\alpha}{}^* I' : \Delta : Q$$

where I' is a suffix of I. Furthermore, Q can be transformed using only the structural rules to the form $\nu\vec{r}.(N \mid C)$, where N is a terminal value (Def. 3.5), and C is a number (possibly zero) of parallel passive containers. The restrictions encoded by $\vec{r}$ cover all passive containers in C.

Proof. (Sketch) By definition 4.13, our proof obligation reduces to establishing that Q can be transformed into the required $\nu\vec{r}.(N \mid C)$ form. By inspection of the structural rules, we see that the rule *EXTRUDE* can be repeatedly used to move restrictions to the top, obtaining the required form. (*ALPHA* rules can

be used to resolve naming conflicts, if any.) To see the correspondence between restrictions and the passive containers, just notice that they are introduced together by *NEWIO*, they are never removed, and all rules respect the scoping of ν bindings. □

Observation 4.16. Note that derivations apply to both pure and IO terms. A derivation either diverges, or ends up with an abstraction or a saturated constructor application for a pure term, or with a term of the form *return* M for an IO term.

Proposition 4.17. *(Derivations for IO-terms in contexts) Let* $I : \Gamma : \nu\vec{r}.(\mathrm{E}[M\,] \mid C)$ *be a closed program state, where* M *is an IO-term. The derivation starting at this state will either diverge, or take the form:*

$$\begin{aligned} I : \Gamma : \nu\vec{r}.(\mathrm{E}[M\,] \mid C) \quad &\xrightarrow{\alpha}{}^{*} \quad I' : \Delta : \nu\vec{r'}.(\mathrm{E}[\mathit{return}\ \ N\,] \mid C') \\ &\xrightarrow{\beta}{}^{*} \quad I'' : \Theta : \nu\vec{r''}.(\mathit{return}\ \ O\ \ \mid C'') \end{aligned}$$

where I' *is a suffix of* I*, and* I'' *is a suffix of* I'*.*

Proof. By inspection of our rules, we see that if the derivation for $\Gamma : \nu\vec{r}.(E[M\,] \mid C)$ terminates, then so must the derivation for $\Gamma : \nu\vec{r}.(M \mid C)$. Hence, by the previous lemma, it must do so in the required intermediate form. The form of the final state is again guaranteed by the previous lemma. □

To be able to talk about strictness (Eq. (1)), we need to identify what $\bot$ means for the type $IO\ \tau$:

Definition 4.18. (*Silent derivations*) A derivation is silent if its effect is empty.

Definition 4.19. (*Bottoms of IO*) A closed program state $(I : \Gamma : M) :: IO\ \tau$ is a bottom element ($\bot$) for the type $IO\ \tau$, iff the derivation for $I : \Gamma : M$ silently diverges.

Example 4.20. It is easy to see that Program State (5) is not a $\bot$ of IO, but Program State (6) is. While they both diverge, the former is not silent.

Definition 4.21. (*Strict functions*) Let Γ be a heap and M be a term such that the program state $([\,] : \Gamma : M) :: \tau \to IO\ \sigma$ is closed. M is strict, if, for all I and $\Delta \supseteq \Gamma/M$, $x \notin bn(\Gamma)$, the derivation for

$$I : (\Delta, x \mapsto \bullet) : M\ x$$

is silently divergent.

5. Examples

We revisit the examples given in Section 2, and show how our semantics can handle them. In these examples, we will use the letters $a, b, \ldots$ to represent heap variables as well. To save space, we will apply the structural rules silently.

Example 5.1. We will revisit Example 2.1. We first remove the *do* notation in favor of explicit $\gg\!=$'s:

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

To reduce clutter, we will not write the input stream explicitly. We have:

$\{\} : \mathit{fixIO}\ (\lambda cs.\ \mathit{getChar} \gg\!= \lambda c.\ \mathit{return}\ (c : cs))$
$\longrightarrow^*$ (FIXIO - FUN)
$\{z \mapsto \bullet,\ a \mapsto z\} : \mathit{getChar} \gg\!= \lambda c.\ \mathit{return}\ (c : a) \gg\!= \mathit{update}_z$
$\xrightarrow{?ch}$ (GETC – assume input stream has *ch* in front)
$\{z \mapsto \bullet,\ a \mapsto z\} : \mathit{return}\ \mathit{ch} \gg\!= \lambda c.\ \mathit{return}\ (c : a) \gg\!= \mathit{update}_z$
$\longrightarrow^*$ (LUNIT - FUN)
$\{z \mapsto \bullet,\ a \mapsto z, b \mapsto \mathit{ch}\} : \mathit{return}\ (b : a) \gg\!= \mathit{update}_z$
$\longrightarrow$ (LUNIT)
$\{z \mapsto \bullet,\ a \mapsto z, b \mapsto \mathit{ch}\} : \mathit{update}_z\ (b{:}a)$
$\longrightarrow$ (UPDATE)
$\{z \mapsto b : a,\ a \mapsto z, b \mapsto \mathit{ch}\} : \mathit{return}\ z.$

The derivation terminates with a terminal program state at this point. Hence the initial program state is normal. The final heap contains the cyclic structure that represents the infinite list of *ch*'s: The character that was read by *getChar*. In case elements of this list are demanded in a context, the usual demand-driven rules modeled by our semantics would let us produce enough elements to satisfy the need. If the input stream is empty to start with, the derivation will simply block at the point where the *GETC* rule is applied, and wait forever, *i.e.*, the derivation will diverge by getting stuck.

Example 5.2. We now reconsider Example 2.2, which involves reference cells. Again, removing *do*-notation and simplifying the patterns, we get:

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

Since there are no calls to *getChar*, the input stream does not matter. That is, we will simply copy the same input stream through all transitions in our derivation. Therefore, we simply do not write it explicitly in what follows.

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

$\{\} : \mathit{fixIO}\ (\lambda t.\ \mathit{new}\ (\mathit{fst}\ t) \gg\!= \lambda y.\mathit{return}\ (1 : \mathit{fst}\ t,\ y))$
$\longrightarrow^*$ (FIXIO - FUN)
$\{z \mapsto \bullet,\ a \mapsto z\} : \mathit{new}\ (\mathit{fst}\ a) \gg\!= \lambda y.\ \mathit{return}\ (1 : \mathit{fst}\ a,\ y) \gg\!= \mathit{update}_z$
$\longrightarrow$ (NEWIO)
$\{z \mapsto \bullet,\ a \mapsto z,\ b \mapsto \mathit{fst}\ a\} :$
$\quad \nu r.(\mathit{return}\ r \gg\!= \lambda y.\mathit{return}\ (1 : \mathit{fst}\ a,\ y) \gg\!= \mathit{update}_z \mid \langle b \rangle_r)$
$\longrightarrow^*$ (LUNIT - FUN)
$\{z \mapsto \bullet,\ a \mapsto z,\ b \mapsto \mathit{fst}\ a,\ c \mapsto r\} :$

$$\nu r.(\mathit{return}\ (1 : \mathit{fst}\ a,\ c) \gg\!= \ \mathit{update}_z\ \mid \langle b\rangle_r)$$
$\longrightarrow^*$ (LUNIT - UPDATE)
$$\{z \mapsto (1 : \mathit{fst}\ a, c),\ a \mapsto z,\ b \mapsto \mathit{fst}\ a,\ c \mapsto r\} : \nu r.(\mathit{return}\ z\ \mid \langle b\rangle_r).$$

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

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

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 $(1 : \mathit{fst}\ a,\ c)$ refers to a function application: the pairing constructor applied to the terms $1 : \mathit{fst}$ 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.

Example 5.3. This example demonstrates the importance of the side condition of the *EXTRUDE* rule. Consider:

$$\begin{aligned}&\mathbf{do}\ j \leftarrow \mathit{new}\ 5\\ &\quad\ k \leftarrow \mathit{new}\ j\\ &\quad\ l \leftarrow \mathit{read}\ k\\ &\quad\ \mathit{read}\ l.\end{aligned}$$

By removing the *do*-notation, we get:

$$\mathit{new}\ 5 \gg\!= \ \mathit{new} \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}.$$

We will try to give a derivation for this expression, ignoring the side condition of the *EXTRUDE* rule. Again the input stream is irrelevant, and hence ignored:

$$\{\} : \mathit{new}\ 5 \gg\!= \ \mathit{new} \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}$$
$\longrightarrow$ (NEWIOREF)
$$\{x \mapsto 5\} : \nu j.(\mathit{return}\ j \gg\!= \ \mathit{new} \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}\ \mid \langle x\rangle_j)$$
$\longrightarrow^*$ (LUNIT-NEWIOREF)
$$\{x \mapsto 5,\ y \mapsto j\} : \nu j.(\nu k.(\mathit{return}\ k \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}\ \mid \langle y\rangle_k) \mid \langle x\rangle_j)$$
$\longrightarrow$ (COMM)
$$\{x \mapsto 5,\ y \mapsto j\} : \nu j.(\langle x\rangle_j \mid \nu k.(\mathit{return}\ k \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}\ \mid \langle y\rangle_k))$$

$\longrightarrow$ (EXTRUDE – *incorrect application*)
$\{x \mapsto 5,\ y \mapsto j\} : \nu j.(\langle x \rangle_j) \mid \nu k.(\mathit{return}\ k \gg\!= \ \mathit{read} \gg\!= \ \mathit{read} \mid \langle y \rangle_k)$
$\longrightarrow^*$ (LUNIT - READ - LUNIT)
$\{x \mapsto 5,\ y \mapsto j\} : \nu j.(\langle x \rangle_j) \mid \nu k.(\mathit{read}\ y \mid \langle y \rangle_k)$
$\longrightarrow$ (READIOEVAL)
$\{x \mapsto 5,\ y \mapsto j\} : \nu j.(\langle x \rangle_j) \mid \nu k.(\mathit{read}\ j \mid \langle y \rangle_k).$

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 us construct the slice when the rule is applied:

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

By equation 4, the slice is: $\{y \mapsto j\}$. Since $j \in \mathit{fn}(\{y \mapsto j\})$, *EXTRUDE* is not applicable. The side condition prevents the creation of the dangling reference.

6. Properties of *fixIO*

Equipped with the semantics we have presented so far, we are now in a position to look at the properties of *fixIO*.

6.1. Strictness

Consider Equation (1), and let Γ be a heap where f is properly bound. Assuming f is strict (Def. 4.21), we will have:

$$I : \Gamma : \mathit{fixIO}\ f \ \longrightarrow I : (\Gamma,\ z \mapsto \bullet) : f\ z \gg\!= \ \mathit{update}_z$$

by a single application of the *FIXIO* rule. The current context specifies that the application $f\ z$ should be evaluated. By Definition 4.21, the derivation will silently diverge. But then, by Definition 4.19, this divergence implies that *fixIO* f is $\bot$.

To illustrate the notion of strictness for IO computations, we will consider several examples.

Example 6.1. Using **if** as a shorthand for **case** over the boolean type, consider:

$I : \{\} : \mathit{fixIO}\ (\lambda x.\ \mathbf{if}\ x == 0\ \mathbf{then}\ \mathit{return}\ 1\ \mathbf{else}\ \mathit{return}\ 2)$
$\longrightarrow$ (FIXIO - FUN)
$I : \{z \mapsto \bullet,\ a \mapsto z\} : \mathbf{if}\ a = 0\ \mathbf{then}\ \mathit{return}\ 1\ \mathbf{else}\ \mathit{return}\ 2 \gg\!= \ \mathit{update}_z$
... detected black hole ...

In the last step, the *FUN* rule is not applicable because there are no reductions for the current term in the functional layer.

Example 6.2. Consider the following 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:

$$
\begin{array}{l}
\quad I : \{\} : \mathit{fixIO}\ (\lambda x.\ \mathit{return}\ x) \gg\!= \ \mathit{putChar} \\
\longrightarrow \text{(FIXIO - FUN)} \\
\quad I : \{z \mapsto \bullet,\ a \mapsto z\} : \mathit{return}\ a \gg\!= \ \mathit{update}_z \gg\!= \ \mathit{putChar} \\
\longrightarrow \text{(LUNIT)} \\
\quad I : \{z \mapsto \bullet,\ a \mapsto z\} : \mathit{update}_z\ a \gg\!= \ \mathit{putChar} \\
\longrightarrow^{*} \text{(UPDATE - LUNIT)} \\
\quad I : \{z \mapsto a,\ a \mapsto z\} : \mathit{putChar}\ z \\
\ldots\ \mathit{detected\ black\ hole}\ \ldots
\end{array}
$$

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

Example 6.3. Consider the function:

$$\lambda a.\ \mathit{putChar}\ \texttt{'q'} \gg \textbf{if}\ a == 1\ \textbf{then}\ \mathit{return}\ 1\ \textbf{else}\ \mathit{return}\ 2$$

which is not strict according to our semantics. Here is the derivation for it:

$$
\begin{array}{l}
\quad I : \{\} : \mathit{fixIO}\ (\lambda a.\ \mathit{putChar}\ \texttt{'q'} \gg \textbf{if}\ a == 1\ \textbf{then}\ \mathit{return}\ 1\ \textbf{else}\ \mathit{return}\ 2) \\
\longrightarrow^{*} \text{(FIXIO - FUN)} \\
\quad I : \{z \mapsto \bullet,\ a \mapsto z\} : \\
\qquad \mathit{putChar}\ \texttt{'q'} \gg \textbf{if}\ a == 1\ \textbf{then}\ \mathit{return}\ 1\ \textbf{else}\ \mathit{return}\ 2 \gg\!= \ \mathit{update}_z \\
\xrightarrow{\texttt{!q}} \text{(PUTC)} \\
\quad I : \{z \mapsto \bullet,\ a \mapsto z\} : \textbf{if}\ a == 1\ \textbf{then}\ \mathit{return}\ 1\ \textbf{else}\ \mathit{return}\ 2 \gg\!= \ \mathit{update}_z \\
\ldots\ \mathit{detected\ black\ hole}\ \ldots
\end{array}
$$

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

6.2. Purity

Consider Equation (2), where we will use a **let** expression to capture the functionality of *fix*:

$$\mathit{fixIO}\ (\mathit{return} \cdot h) = \mathit{return}\ (\textbf{let}\ a = h\ a\ \textbf{in}\ a).$$

Assume Γ is a heap such that $([\,] : \Gamma : h) :: \tau \to \tau$. On the left hand side, we have:

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

Considering the right-hand-side, we immediately get:

$$I : \Gamma : \mathit{return}\ (\mathbf{let}\ a\ =\ h\ a\ \mathbf{in}\ a).$$

We should now prove that these two program states are equivalent, *i.e.*, that the rules in our system cannot tell them apart. Such an argument would require a notion of program state equivalence that is more general than what our structural rules provide. Intuitively, the program states above will be considered equivalent if we can show that

$$I : (\Gamma,\ z \mapsto h\ a,\ a \mapsto z) : z \quad \equiv \quad I : \Gamma : \mathbf{let}\ a\ =\ h\ a\ \mathbf{in}\ a.$$

Note that the second program state reduces to $I : (\Gamma,\ z \mapsto h\ z) : z$. Hence, the equivalence is clear provided we adopt a compaction rule that gets rid of the indirection *via* a in the first heap. To formalize this argument, we need a precise definition of program state equivalence and a proof system for showing when two program states are the same. Development of a such a system is beyond the scope of the current paper.

6.3. Left shrinking

We will now consider equation (3). Assume q is bound appropriately in the heap Γ. For the left hand side we get:

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

On the right hand side, we have $I : \Gamma : q \gg\!= \lambda y.\ \mathit{fixIO}\ (\lambda x.\ f\ x\ y)$. Now, if the derivation for q diverges, both derivations will diverge in the exact same way, that is both sides are equivalent. Otherwise, by Lemma 4.15, we will have:

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

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 exactly the same way, we do not show them explicitly in the following discussion. Using the *HEAPEXT* and *EXTRUDE* rules silently, the left hand side yields:

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

Let us look at the right hand side:

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

Hence, the left shrinking property holds for *fixIO*. We conclude that, with respect to our semantics, *fixIO* is a value recursion operator for the IO monad.

7. Conclusions And Future Work

We have described a semantics for *fixIO*, and shown that it satisfies the axioms required for value recursion operators [5, 6]. 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 I/O, mutable variables, and value recursion;
- 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 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 Peyton Jones's work [26]. This extension does not present any technical challenges. The difficulty, however, lies in extending the approach with asynchronous exceptions [19]. 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.

Developing proof techniques for establishing program equivalence remains as a challenging task. Although reasoning techniques are quite well developed for the functional sublanguage, we do not have very strong tools to deal with monadic effects and mutable variables. Needless to say, the same goes for the Haskell language in general. Although the system we have described can be used to give semantics to concrete examples, it is harder to use it when reasoning about symbolic terms.

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

References

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

[2] Zena M. Ariola and Stefan Blom. Cyclic lambda calculi. In *Theoretical Aspects of Computer Software*, pages 77–106, 1997.

[3] Nick Benton and Martin Hyland. Traced premonoidal categories (Extended Abstract). In *Fixed Points in Computer Science Workshop, FICS'02*, 2002.

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, Baltimore, July 1998.

[5] Levent Erkök. *Value recursion in monadic computations*. PhD thesis, OGI School of Science and Engineering, OHSU, Portland, Oregon, 2002.

[6] Levent Erkök and John Launchbury. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 174–185. ACM Press, September 2000.

[7] Levent Erkök, John Launchbury, and Andrew Moran. Semantics of *fixIO*. In *Fixed Points in Computer Science Workshop, FICS'01*, September 2001.

[8] Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[9] A. D. Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, September 1994.

[10] Masahito Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *Typed Lambda Calculus and Applications*, pages 196–213, 1997.

[11] Masahito Hasegawa. *Models of Sharing Graphs, A categorical semantics of let and letrec*. Distinguished Dissertations in Computer Science. Springer Verlag, 1999.

[12] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, May 2000.

[13] Alan Jeffrey. Premonoidal categories and a graphical view of programs. Unpublished manuscript. URL: `fpl.cs.depaul.edu/ajeffrey/premon/paper.html`, 1997.

[14] Mark P. Jones. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*, 1999.

[15] A. Joyal, R. H. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

[16] John Launchbury. 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*, pages 144–154, 1993.

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

[18] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[19] Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 20–22 2001.

[20] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

[21] John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 90–101. IEEE Computer Society Press, 1998.

[22] Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, May 1999.

[23] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

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

[25] Ross Paterson. A new notation for arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming, ICFP'01, Florence, Italy*, pages 229–240. ACM Press, September 2001.

[26] Simon L. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, 2001.

[27] Simon L. Peyton Jones and John Hughes. (Editors.) Report on the programming language Haskell 98, a non-strict purely-functional programming language. URL: `www.haskell.org/onlinereport`, February 1999.

[28] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, 1993.

[29] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[30] John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, 1997.

[31] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[32] Alex Simpson and Gordon Plotkin. Complete axioms for categorical fixed-point operators. In *Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science*, pages 30–41, 2000.

.