Traced Premonoidal Categories

Nick Benton
Microsoft Research
Cambridge, UK
nick@microsoft.com

Martin Hyland
University of Cambridge
Department of Pure Mathematics
and Mathematical Statistics
M.Hyland@dpmms.cam.ac.uk

Abstract

Motivated by some examples from functional programming, we propose a generalisation of the notion of trace to symmetric premonoidal categories and of Conway operators to Freyd categories. We show that, in a Freyd category, these notions are equivalent, generalising a well-known theorem of Hasegawa and Hyland.

1 Introduction

Monads were introduced into computer science by Moggi [18] as a structuring device in denotational semantics and soon became a popular abstraction for writing actual programs, particularly for expressing and controlling side-effects in 'pure' functional programming languages such as Haskell [25, 17]. Power and Robinson subsequently introduced *premonoidal categories* as a generalisation of Moggi's computational models [21], whilst Hughes developed *arrows*, which are the equivalent programming abstraction [12].

Some uses of monads in functional programming seem to call for a kind of recursion operator on computations for which, informally, the recursion 'only takes place over the values'. For example, the Haskell Prelude defines the (internally implemented) `ST` and `IO` monads for, respectively, potentially state-manipulating and input/output-performing computations. These come equipped with polymorphic functions

```
fixST :: (a -> ST a) -> ST a
fixIO :: (a -> IO a) -> IO a
```

which allow computations to be recursively defined in terms of the values they produce. For example, the following program uses `fixIO`[1] to extend a cunning cyclic programming trick due to Bird [1] to the case of side-effecting computations. `replacemin` computes a tree in which every leaf of the argument has been replaced by the minimum of all the leaves. It does this in a single pass over the input and prints out each leaf as it encounters it:[2]

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

f :: Tree Int -> Int -> IO (Int,Tree Int)
f (Leaf n) m = do print n
                  return (n, Leaf m)
f (Branch t1 t2) m =
    do (m1,r1) <- f t1 m
       (m2,r2) <- f t2 m
       return (min m1 m2, Branch r1 r2)

replacemin :: Tree Int -> IO (Int, Tree Int)
-- m is argument to and part of the result of f
replacemin t = fixIO (\ ~(m,r) -> f t m)
```

As another (though still somewhat contrived) example, consider modelling the heap of a fictitious pure Scheme-like language at a fairly low level. One might interpret heap-manipulating computations using a monad `T` which is an instance of a type class something like this

```
class Monad T => HeapMonad T where
  alloc  :: (Int, Int) -> T Int
  lookup :: Int -> T (Int,Int)
  free   :: Int -> T ()
```

The intention is that `alloc` takes two integers and returns a computation which finds a free cons cell in the heap, fills it with those two integers and returns the (strictly positive) address of the allocated cell. `lookup` takes an integer address and returns the contents of that cons cell, whilst `free` marks a particular address as available for future allocations. Since the values in the *car* and *cdr* of cells can be used as the addresses of other cells, we can intepret programs which build data structures such as lists in the heap. What if the language we are interpreting can create cyclic structures (for example, closures for recursive functions)? At the machine level, cyclic structures are created by allocating cells containing dummy values and then 'tying the knot' by overwriting those dummy values with the addresses returned by the allocator. Hence we could just provide destructive update operations

```
setcar :: Int -> Int -> T ()
setcdr :: Int -> Int -> T ()
```

and use those to create cycles. However, if the interpreted language itself does not include destructive assignment, but only creates cycles using higher-level constructs, then adding assignment operations to the monad breaks an abstraction barrier. One solution is to add a recursion operation to the monad

[1] We should note that `fixIO` does not *actually* satisfy the axioms we will propose. However, the basic pattern would remain the same, though the code would be a little longer, if we had performed side-effects involving state instead.

[2] The tilde `~` on the last line specifies 'lazy' pattern matching for the pair `(m,r)`. Haskell's tuples are actually lifted products and pattern matching is, by default, strict. Without the tilde the function would diverge.

```
fixT :: (a -> T a) -> T a
```

with a definition such that the following code creates a two-element cyclic list (and returns the addresses of both cells):

```
onetwocycle :: T (int,int)
onetwocycle  = fixT (\~(x,y)->
                       do { x' <- alloc(1,y)
                            y' <- alloc(2,x)
                            return (x',y')
                          })
```

Observe that although the computation is recursively defined, it should only perform the two allocation side-effects once.

Many of the real uses of this kind of recursion have the flavour of the previous example: they involve computations which create cyclic structures for which the identity, order of creation or multiplicity of creation of the objects in the structure is significant. An interesting example arises in work on using Haskell to model hardware. Early versions of both Lava [2] and Hawk [16] specified circuits in a monadic style, instantiating the monad differently for different applications (such simulating the circuit, generating a netlist or interfacing with a theorem prover). Cyclic circuits (i.e. those with feedback) were defined in essentially the style used to define `onetwocycle` above. Lava has moved away from that style, in part because it is syntactically awkward.[3] Launchbury et al. [16] also noted that programming in a monadic style with `fixT` is awkward, and suggested extending Haskell's `do` notation to allow recursive bindings. That suggestion was followed up by Launchbury and Erkök, who proposed an axiomatisation of operators like `fixT` (which they call `mfix`) and showed how the `do` notation can be extended to allow recursive bindings in the case that the underlying monad supports such an `mfix` operation [15].

Launchbury and Erkök's axiomatisation of `mfix` is partly in terms of equations and partly in terms of inequations, intended to be interpreted in the 'usual' (slightly informal) concrete domain theoretic model of Haskell. One striking feature of [15] is that it does not appear to build on any of the large body of existing work on axiomatic/categorical treatments of recursion, even those (such as [7]) which consider fixed points in terms of monads. The authors cite some of this work but state, quite correctly, that the non-standard kind of recursion in which they are interested is different from that covered in the literature. Although the presence of a fixpoint object [7], for example, allows an operator with the same type as `mfix` to be defined, it is not of the kind we want.

From a categorical perspective, we seem to want a notion of recursion or feedback on the Kleisli category of a CCC with a strong monad. There is a special case of this situation in which earlier work *does* provide an answer. Although none of Launchbury and Erkök's examples are of commutative monads, in that case the Kleisli category will be symmetric monoidal and Joyal, Street and Verity's notion of *trace* seems to fit the bill [14].

In the general case of a non-commutative monad, however, the Kleisli category will only be symmetric *pre*monoidal. The work described here grew firstly from the natural mathematical question of what the right definition of traced premonoidal category might be, and secondly from wondering whether an answer might provide a sensible categorical semantics for the kind of fixpoint operators described in [15]. We give a natural, straightforward and well-behaved answer to the first question, though it only accounts for a rather special subset of the cases considered by Launchbury and Erkök.

[3] Lava now uses a modified version of Haskell with 'observable sharing': allowing new name generation as an implicit side-effect of every expression and hence changing the equational theory of the language [6].

2 Background

2.1 Premonoidal Categories

For a careful definition of the notion of *(symmetric) premonoidal category* and *(symmetric) premonoidal functor*, see Power and Robinson's paper [21]. Briefly, a premonoidal category is a monoidal category except that the tensor product $\otimes$ need only be a functor in each of the two variables separately. Thus if $f : A \to B$ and $g : A' \to B'$ in a premonoidal category $\mathbb{K}$ then the two evident morphisms $A \otimes A' \to B \otimes B'$

$$\begin{aligned} f \ltimes g &= A \otimes A' \xrightarrow{f \otimes A'} B \otimes A' \xrightarrow{B \otimes g} B \otimes B' \\ f \rtimes g &= A \otimes A' \xrightarrow{A \otimes g} A \otimes B' \xrightarrow{f \otimes B'} B \otimes B' \end{aligned}$$

are not generally equal.

We generally write I for the unit of the tensor in a (pre)monoidal category, σ for the symmetry if there is one, and λ, ρ, α for the natural isomorphisms

$$\begin{aligned} \lambda &: I \otimes A \to A \\ \rho &: A \otimes I \to A \\ \alpha &: (A \otimes B) \otimes C \to A \otimes (B \otimes C) \end{aligned}$$

However, since we have coherence theorems for (symmetric) (pre)monoidal categories [21], we will usually elide the structural isomorphisms.

Definition 2.1. A morphism $f : A \to B$ in a premonoidal category $\mathbb{K}$ is *central* if for all $g : A' \to B'$ in $\mathbb{K}$, $f \ltimes g = f \rtimes g$. If at least one of f and g is central, then we may unambiguously write $f \otimes g$. The *centre* $Z(\mathbb{K})$ of a premonoidal category $\mathbb{K}$ is the monoidal subcategory of $\mathbb{K}$ with the same objects but only the central morphisms.

The inclusion functor $Z(\mathbb{K}) \to \mathbb{K}$ is a strict, identity-on-objects premonoidal functor (and symmetric if $\mathbb{K}$ is). In more recent work Power in particular has stressed the importance from the algebraic point of view in having an explicit choice of centre. That is, one is interested in the situation where one has a functor $J : \mathbb{M} \to \mathbb{K}$ from a specified (symmetric) monoidal subcategory of a (symmetric) premonoidal $\mathbb{K}$; J factors through $Z(\mathbb{K})$, so this amounts to specifying a particular subcategory of central morphisms. (For many results J does not even need to be faithful, but we do not consider that generality here.) We call a $J : \mathbb{M} \to \mathbb{K}$ as above a *centred premonoidal category*, but since this is our preferred notion we usually drop the 'centred'. In this context, by *central* morphisms we shall mean the morphisms of $\mathbb{M}$. One should think of $\mathbb{M}$ as a category of *values* and $\mathbb{K}$ as a category of possibly-effectful *computations*. An important special case is the following:

Definition 2.2. A *Freyd category* [22] is specified by a cartesian category $\mathbb{C}$, a symmetric premonoidal category $\mathbb{K}$ and an identity-on-objects strict symmetric premonoidal functor $J : \mathbb{C} \to \mathbb{K}$.

Note that morphisms in the specified centre of a Freyd category are 'pure' not merely in the sense of commuting with arbitrary effectful computations, but also in being copyable and discardable.

Example 2.1. If T is a strong monad on a symmetric monoidal category $\mathbb{M}$, then the Kleisli category $\mathbb{M}_T$ is symmetric premonoidal and the canonical functor from $\mathbb{M}$ to $\mathbb{M}_T$ is strict symmetric premonoidal. Thus in the case that $\mathbb{M}$ is cartesian, we have a Freyd category. If the monad is commutative, then $\mathbb{M}_T$ is symmetric monoidal and $J : \mathbb{M} \to \mathbb{M}_T$ is strict symmetric monoidal.

2.2 Traces and Fixpoints

The notion of traced monoidal category was introduced in [14]. The use of traces to interpret recursion in programming languages and the relationship between traces and fixpoints have attracted much attention in recent years, beginning with Hasegawa's thesis [11]. Categorical axiomatisations of fixpoint operators have been extensively studied, see [7, 19, 4] for example; a particularly crisp and up-to-date account appears in [24].

Definition 2.3. A *trace* on a symmetric monoidal category $(\mathbb{M}, \otimes, I, \lambda, \rho, \alpha, \sigma)$ is a family of functions

$$\mathrm{tr}^U_{A,B} : \mathbb{M}(A \otimes U, B \otimes U) \to \mathbb{M}(A, B)$$

satisfying the following conditions

- Naturality in A (Left Tightening).
  If $f : A' \otimes U \to B \otimes U$, $g : A \to A'$ then
  $$\mathrm{tr}^U_{A,B}((g \otimes U); f) = g; \mathrm{tr}^U_{A',B}(f) : A \to B$$
- Naturality in B (Right Tightening).
  If $f : A \otimes U \to B' \otimes U$, $g : B' \to B$ then
  $$\mathrm{tr}^U_{A,B}(f; (g \otimes U)) = \mathrm{tr}^U_{A,B'}(f); g : A \to B$$
- Dinaturality (Sliding).
  If $f : A \otimes U \to B \otimes V$, $g : V \to U$ then
  $$\mathrm{tr}^U_{A,B}(f; (B \otimes g)) = \mathrm{tr}^V_{A,B}((A \otimes g); f) : A \to B$$
- Action (Vanishing). If $f : A \to B$ then
  $$\mathrm{tr}^I_{A,B}(\rho; f; \rho^{-1}) = f : A \to B$$
  and if $f : A \otimes (U \otimes V) \to B \otimes (U \otimes V)$ then
  $$\mathrm{tr}^{U \otimes V}_{A,B}(f) = \mathrm{tr}^U_{A,B}(\mathrm{tr}^V_{A \otimes U, B \otimes U}(\alpha; f; \alpha^{-1}))$$
- Superposing. If $f : A \otimes U \to B \otimes U$ then
  $$\mathrm{tr}^U_{C \otimes A, C \otimes B}(\alpha; C \otimes f; \alpha^{-1}) = C \otimes \mathrm{tr}^U_{A,B}(f) \ : \ C \otimes A \to C \otimes B$$
- Yanking. For all U, $\mathrm{tr}^U_{U,U}(\sigma_{U,U}) = U : U \to U$.

Monoidal categories provide a formal basis for reasoning about many of the graphical 'boxes and wires' notations used in computer science. The trace axioms are presented graphically in Figure 1, though we do not consider the formal semantics of such diagrams here.

Definition 2.4. A *parameterized fixpoint operator* on a cartesian category $\mathbb{C}$ is a family of functions

$$(\cdot)^\dagger : \mathbb{C}(A \times U, U) \to \mathbb{C}(A, U)$$

satisfying

- Naturality. If $f : B \times U \to U$ and $g : A \to B$ then
  $$g; f^\dagger = ((g \times U); f)^\dagger : A \to U$$
- Fixed point property. If $f : A \times U \to U$ then
  $$\langle A, f^\dagger \rangle; f = f^\dagger : A \to U$$

The above definition is rather weak. Well-behaved fixpoint operators typically satisfy other interesting conditions.

Definition 2.5. A *Conway operator* is a parameterized fixpoint operator which additionally satisfies

- Parameterized Dinaturality. If $f : A \times V \to U$ and $g : A \times U \to V$ then
  $$\langle A, (\langle \pi_1, f \rangle; g)^\dagger \rangle; f = (\langle \pi_1, g \rangle; f)^\dagger : A \to U$$
- Diagonal Property. If $f : A \times U \times U \to U$ then
  $$((A \times \Delta); f)^\dagger = (f^\dagger)^\dagger : A \to U$$

Parameterized dinaturality is easily seen to imply the parameterized fixed point property and, in some concrete categories of domains, is sufficient to characterize the least fixed point operator uniquely [23]. Conway operators satisfy various other useful identities, including the 'Bekič property', which allows simultaneous fixed points to be reduced to sequential ones.

There are also further 'uniformity' properties which a fixpoint operator may have [24], but we shall not consider those in the present paper.

An important theorem about traces and fixpoints is the following, which is due (independently) to Hasegawa and to Hyland, though its essential combinatorial content had been observed earlier in a slightly different context [3, 5]:

Theorem 2.1 (Hasegawa, Hyland). *To give a trace on a cartesian category $\mathbb{C}$ is to give a Conway operator on $\mathbb{C}$.* □

3 Traces and Fixpoint Operators on Premonoidal Categories

So, what is an appropriate generalisation of the notions of trace and fixpoint operator to the premonoidal case? We want definitions which make sense, have useful concrete instances, give the monoidal versions as special cases and lead to a generalisation of Theorem 2.1.

3.1 Symmetric Premonoidal Traces

We start by trying to generalise the definition of trace to a centred symmetric premonoidal category $J : \mathbb{M} \to \mathbb{K}$. Although none of the conditions in Definition 2.3 are expressed in terms of tensoring arbitrary morphisms (in which case we'd certainly have to reexamine them), we cannot simply leave the definition unchanged:

Figure 1: Trace Axioms

Proposition 3.0.1. *A symmetric premonoidal category with a trace as defined in Definition 2.3 is actually monoidal.* □

The key step in the proof of the previous Proposition uses the Sliding axiom to commute the side-effects of two computations. This observation motivates the following definition:

Definition 3.1. A *trace* on a centred symmetric premonoidal category $J : \mathbb{M} \to \mathbb{K}$ is is a family of functions

$$\mathrm{tr}^U_{A,B} : \mathbb{K}(A \otimes U, B \otimes U) \to \mathbb{K}(A, B)$$

satisfying the same conditions given in Definition 2.3 *except* that the Sliding axiom is replaced by

- Premonoidal Sliding. If $f : A \otimes U \to B \otimes V$ and if $g : V \to U$ is a central morphism then

$$\mathrm{tr}^U_{A,B}(f; (B \otimes g)) = \mathrm{tr}^V_{A,B}((A \otimes g); f) : A \to B$$

and we impose the further requirement

- Centre Preservation. If $f : A \otimes U \to B \otimes U$ is central then so is $\mathrm{tr}^U_{A,B} f : A \to B$.

Clearly, if $J : \mathbb{M} \to \mathbb{K}$ has a premonoidal trace on $\mathbb{K}$, the restriction of that trace to $\mathbb{M}$ is a trace operator in the traditional sense of Definition 2.3. In particular, Definition 3.1 really is a generalisation of Definition 2.3.

Requiring the trace to preserve the distinguished centre $\mathbb{M}$ is largely a matter of taste: we prefer to keep our equations algebraic. Even without the condition it is still easy to see that the trace preserves $Z(\mathbb{K})$:

Proposition 3.0.2. *If $f : A \otimes U \to B \otimes U$ is in $Z(\mathbb{K})$ and $g : C \to D$ then $g \ltimes \mathrm{tr}^U_{A,B}(f) = g \rtimes \mathrm{tr}^U_{A,B}(f)$.* □

It might also be remarked that the premonoidal sliding condition appears somewhat asymmetric, since it requires that g, rather than one of f and g, be central. However, a little calculation shows that the symmetric case is a consequence:

Proposition 3.0.3. *Assume $f : A \otimes U \to B \otimes V$ is central and $g : V \to U$, then $\mathrm{tr}^V_{A,B}((A \otimes g); f) = \mathrm{tr}^U_{A,B}(f; B \otimes g)$.* □

3.2 Symmetric Premonoidal Fixpoints

We now turn to generalising the notion of fixpoint operator to the premonoidal case. Since some of the axioms involve duplication and discarding, we will assume that we are working in a Freyd category $J : \mathbb{C} \to \mathbb{K}$. We also use Δ, π_1, $\langle \cdot, \cdot \rangle$, etc. as shorthand notation for the lifting of the appropriate operations from $\mathbb{C}$ to $\mathbb{K}$ (i.e. we elide uses of J). The notation $\langle f, g \rangle$ is ambiguous unless we specify the order in which the components are computed, but we shall only use it in the case one of the maps is central.

Definition 3.2. A *parameterized fixpoint operator* on a Freyd category $J : \mathbb{C} \to \mathbb{K}$ is a family of functions

$$(\cdot)^* : \mathbb{K}(A \otimes U, U) \to \mathbb{K}(A, U)$$

which satisfies

- Centre Preservation. If $f : A \otimes U \to U$ is central then so is $f^* : A \to U$.

- Naturality. If $f : B \otimes U \to U$ and $g : A \to B$ then

$$g; f^* = ((g \otimes U); f)^* : A \to U$$

- Central Fixed Point Property. If $f : A \otimes U \to U$ is central, then

$$\langle A, f^* \rangle; f = f^* : A \to U$$

Just as in the cartesian case, this is the bare minimum one might require of a fixpoint operator. We are interested in rather stronger conditions, and propose the following as an appropriate generalisation of Conway operators on cartesian categories:

Definition 3.3. A parameterized fixpoint operator $(\cdot)^*$ on a Freyd category is a *Conway operator* if it satisfies the following conditions:

1. *Parallel Property.* If $f : A \otimes U \to U$ and $g : B \otimes V \to V$ with one of f and g central then

$$(A \otimes \sigma \otimes V \,;\, f \otimes g)^* = f^* \otimes g^* \;:\; A \otimes B \to U \otimes V$$

2. *Withering Property.* If $f : A \otimes U \to B \otimes U$ and $g : B \to C$ then

$$(\langle \pi_1, \pi_3 \rangle; f; g \otimes U)^* = (\langle \pi_1, \pi_3 \rangle; f)^*; g \otimes U \;:\; A \to C \otimes U$$

3. *Diagonal Property.* If $f : A \otimes U \otimes U \to U$ then

$$((A \otimes \Delta); f)^* = (f^*)^* : A \to U$$

The axioms of a premonoidal Conway operator are shown graphically in Figure 3.2, where we follow Jeffrey [13] in using a heavy line to indicate the sequencing of effects in $\mathbb{K}$ (and that line runs outside those boxes intended to represent central morphisms). The diagonal property is essentially the same as in the cartesian case, but the parallel and withering properties are more unusual.

There *is* a natural generalization of the dinaturality condition to Freyd categories:

Definition 3.4. A parameterized fixpoint operator $(\cdot)^*$ on a Freyd category satisfies *parameterized central dinaturality* if, given $f : A \otimes U \to V$ and $g : A \otimes V \to U$ with g central

$$(\langle \pi_1, f \rangle; g)^* = \langle A, (\langle \pi_1, g \rangle; f)^* \rangle; g$$

As in the cartesian case, parameterized central dinaturality clearly implies the central fixed point property. But in the case of Freyd categories, the dinaturality condition does not seem sufficient (along with the diagonal property) to establish the equivalence between traces and Conway operators, which is what motivated our parallel and withering axioms. These do imply dinaturality, however:

Proposition 3.0.4. *A Conway operator on a Freyd category satisfies parameterized central dinaturality.* □

Furthermore, our definition of a Conway operator on a Freyd category does generalise the standard one:

Proposition 3.0.5. *Definition 3.3 is equivalent to Definition 2.5 in the case that the category is cartesian.* □

3.3 Relating Fixpoints and Traces in Freyd Categories

We now show our main result: in a Freyd category, to give a premonoidal trace is equivalent to giving a premonoidal Conway operator.

Theorem 3.1. *Let $J : \mathbb{C} \to \mathbb{K}$ be a Freyd category such that $\mathbb{K}$ is traced, as in Definition 3.1. Then the operation*

$$(\cdot)^* : \mathbb{K}(A \otimes U, U) \to \mathbb{K}(A, U)$$

defined by, for $f : A \otimes U \to U$:

$$f^* = \mathrm{tr}^U_{A,U}(f; \Delta)$$

is a Conway operator in the sense of Definition 3.3. □

Remark 3.1. Hasegawa has also given a construction for a fixpoint operator from a trace in the special case of the Kleisli category of a commutative strong monad on a cartesian category (a case in which the premonoidal structure is monoidal) [11, Theorem 7.2.1]. However, restricting our construction to this special case does not generally give the same fixpoint operator. Hasegawa's construction uses the adjunction in an essential way and repeats side-effects.

Theorem 3.2. *Let $J : \mathbb{C} \to \mathbb{K}$ be a Freyd category where $\mathbb{K}$ has a Conway operator $(\cdot)^*$ as defined in Definition 3.3. Then the operation*

$$\mathrm{tr}^U_{A,B}(\cdot) : \mathbb{K}(A \otimes U, B \otimes U) \to \mathbb{K}(A, B)$$

defined by, for $f : A \otimes U \to B \otimes U$

$$\mathrm{tr}^U_{A,B}(f) = (\langle \pi_1, \pi_3 \rangle; f)^*; \pi_1 \;:\; A \to B$$

is a premonoidal trace in the sense of Definition 3.1. □

Proposition 3.2.1. *The constructions of trace from Conway operator and of Conway operator from trace given in Theorems 3.2 and 3.1 respectively are mutually inverse.* □

Thus we have succeeded in establishing a premonoidal generalization of Theorem 2.1.

Remark 3.2. Starting from a fixpoint operator, there is another candidate for the definition of a trace, viz

$$\mathrm{tr}'(f) = \langle A, (f; \pi_2)^* \rangle; f; \pi_1 : A \to B$$

where $f : A \otimes U \to B \otimes U$. If $\mathbb{K}$ is monoidal this is the same as the construction used in Theorem 3.2, but in the general premonoidal case they are different, and tr' does not seem to have useful properties.

4 Examples

4.1 Monoids

Let $\mathbb{M}$ be a traced symmetric monoidal category as in Definition 2.3 and $(M, \mu : M \otimes M \to M, \eta : I \to M)$ be a monoid in $\mathbb{M}$. Let $\mathbb{K}$ be the Kleisli category of the monad $TA = M \otimes A$ on $\mathbb{M}$, so $\mathbb{K}(A, B) = \mathbb{M}(A, M \otimes B)$. Then the tensor on $\mathbb{M}$ lifts so that $J : \mathbb{M} \to \mathbb{K}$ is a centred symmetric premonoidal category (it is monoidal iff M is a commutative monoid).

Figure 2: Premonoidal Conway Axioms

Proposition 4.0.2. *In the above situation, the operation*

$$\hat{\mathrm{tr}}^U_{A,B} : \mathbb{K}(A \otimes U, B \otimes U) \to \mathbb{K}(A, B)$$

defined by $\hat{\mathrm{tr}}^U_{A,B}(f) = \mathrm{tr}^U_{A,M\otimes B}(f)$ *is a premonoidal trace.* □

Notions of computation based on monoids are fairly common. Commutative monoids such as the natural numbers under addition can be used for modelling resource usage (e.g. timed computations) whereas non-commutative monoids model, for example, side-effecting output. In Haskell syntax, the signature could look like this:

```
class Monoid m where
   mult :: (m,m) -> m
   unit :: m

newtype Cross m a = Cross (m,a) deriving Show

instance Monoid m => Monad (Cross m) where
   return a = Cross (unit,a)
   Cross(m,a) >>= f = let Cross(m',b) = f a
                      in Cross(mult (m,m'), b)

instance Monoid [a] where
   mult (s,t) = s ++ t
   unit = []

-- command which writes to the output
output s = Cross(s,())
```

If we then apply our construction of a premonoidal Conway operator from the trace defined in Proposition 4.0.2 then we end up with an `mfix` operation of the type described by Launchbury and Erkök:

```
instance Monoid m => MonadRec (Cross m) where
   mfix f = let Cross(m,a) = f a
            in Cross(m,a)
```

And this does have the expected behaviour:

```
nats_output =
   mfix (\ys -> do output "first "
                   output "second."
                   return (0 : map succ ys))

> nats_output
Cross ("first second.",[0,1,2,3,4,5,6,7,8,9,...
```

The two side effects have happened once only and in the order specified.

4.2 State

Let $\mathbb{M}$ be a traced symmetric monoidal category, S be an object of $\mathbb{M}$ and $\mathbb{K}$ be the category with the same objects as $\mathbb{M}$ and $\mathbb{K}(A, B) = \mathbb{M}(S \otimes A, S \otimes B)$ with the evident composition. If $\mathbb{M}$ is closed then $\mathbb{K}$ is equivalent to the Kleisli category of the state monad $TA = S \multimap S \otimes A$. Then $J : \mathbb{M} \to \mathbb{K}$ is premonoidal.

Proposition 4.0.3. *In the above situation, the operation*

$$\hat{\mathrm{tr}}^U_{A,B} : \mathbb{K}(A \otimes U, B \otimes U) \to \mathbb{K}(A, B)$$

defined by $\hat{\mathrm{tr}}^U_{A,B}(f) = \mathrm{tr}^U_{S\otimes A, S\otimes B}(f)$ *is a premonoidal trace.* □

Again, the derived fixed point operator on the Kleisli category is easily defined in Haskell:

```
newtype State s a = State (s -> (s,a))

instance Monad (State s) where
  return a = State (\s ->(s,a))
  State m >>= f = State (\s -> let (s',a) = m s
                                   State m' = f a
                               in m' s')

instance MonadRec (State s) where
  mfix f = State (\s -> let State m = f a
                            (s',a) = m s
                        in (s',a))
```

Note how the final value, `a` is recursively defined, but the final state `s'` is not – operationally, each time we go around the loop, the initial state is 'snapped back' to `s`.

5 Related Work

Compared with Launchbury and Erkök's work on axiomatising `mfix`, our definitions and results are in a rather more general setting, but account for rather fewer concrete examples. The axioms in [15] are almost identical to our definition of a premonoidal Conway operator except that they weaken some of our equations to inequations (interpreted in a concrete category of domains), add a strictness condition on one and regard some as additional properties which may hold in some cases (i.e. not part of the basic definition of what they call a 'recursive monad'). These weaker conditions admit definitions of `mfix` for monads such as `Maybe` $(1 + (\cdot))$, lazy lists and Haskell's `IO` monad [8] which do not have Conway operators satisfying our conditions.

Paterson has designed a convenient syntax for programming with Hughes's arrows (just as Haskell adds `do` to simplify programming with monads). Paterson's recent paper [20] gives axioms for an `ArrowLoop` operation which are the same as our definition of a premonoidal trace; our results thus prove an equivalence between `ArrowLoop` and a particular (newly identified) class of `mfix`s.

Jeffrey [13] has also considered a variant of traces in a premonoidal setting, though his construction is rather different from ours: he considers a *partial* trace (only applicable to certain maps) on the category of values rather than on that of computations.

Friedman and Sabry have also recently looked at defining an `mfix` operation [9], though their approach is rather different from the axiomatic one which we and the others cited here have taken. They view the ability to define computations recursively as an additional effect and give a 'monad transformer' which adds a state-based updating implementation of recursion to an arbitrary monad. Lifting the operations of the underlying monad to the new one is left to the programmer (and can generally be done in different ways).

6 Conclusions and Further Work

We have formulated and proved a natural generalisation of the theorem relating traces and Conway operators to the case of premonoidal categories. This has applications to the semantics of some non-standard recursion and feedback operations on computations which have been found useful in functional programming.

It would be interesting to see if one could explain Launchbury and Erkök's weaker axiomatisation in a more general setting. The natural thing to try here is to be more explicit about the presence of an abstract lifting monad, along the lines of [10]. This may also help establish a connection with the partial trace operation used by Jeffrey [13]. We would also like to have some more examples.

We are in the process of investigating the premonoidal version of the 'geometry of interaction' construction, which traditionally embeds a traced monoidal category into a compact closed one. This is interesting from a mathematical view and may also have some connection to the building of layered protocol stacks from stateful components.

References

[1] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, 1998.

[3] S. L. Bloom and Z. Esik. Axiomatizing schemes and their behaviors. *Journal of Computer and System Sciences*, 31(3):375–393, 1985.

[4] S. L. Bloom and Z. Esik. *Iteration Theories*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993.

[5] V. E. Cazanescu and Gh. Stefanescu. A general result on abstract flowchart schemes with applications to the study of accessibility, reduction and minimization. *Theoretical Computer Science*, 99:1–63, 1992.

[6] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, 1999.

[7] R. L. Crole and A. M. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98:171–210, 1992.

[8] L. Erkök, J. Launchbury, and A. Moran. Semantics of `fixio`. In *Proceedings of the Workshop on Fixed Points in Computer Science (FICS'01), Firenze, Italy*, September 2001.

[9] D. P. Friedman and A. Sabry. Recursion is a computational effect. Technical Report 546, Computer Science Department, Indiana University, December 2000.

[10] C. Fuhrmann, A. Bucalo, and A. Simpson. An equational notion of lifting monad. *Theoretical Computer Science*, 200? To appear.

[11] M. Hasegawa. *Models of Sharing Graphs (A Categorical Semantics of Let and Letrec)*. Distinguished Dissertations in Computer Science. Springer-Verlag, 1999.

[12] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–112, 2000.

[13] A. Jeffrey. Premondoidal categories and a graphical view of programs. `http://www.cogs.susx.ac.uk/users/alanje/premon/`, June 1998.

[14] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3), 1996.

[15] J. Launchbury and L. Erkök. Recursive monadic bindings. In *International Conference on Functional Programming*, 2000.

[16] J. Launchbury, J. R. Lewis, and B. Cook. On embedding a microarchitectural design language within Haskell. In *International Conference on Functional Programming*, 1999.

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

[18] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[19] P. S. Mulry. Strong monads, algebras and fixed points. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, number 177 in LMS Lecture Notes, pages 202–216, 1992.

[20] R. Paterson. A new notation for arrows. In *Proceedings of the International Conference on Functional Programming*. ACM Press, September 2001.

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

[22] A. J. Power and H. Thielecke. Closed Freyd and κ-categories. In *International Conference on Automata, Languages and Programming*, 1999.

[23] A. K. Simpson. A characterization of the least-fixed-point operator by dinaturality. *Theoretical Computer Science*, 118(2):301–314, 1993.

[24] A. K. Simpson and G. D. Plotkin. Complete axioms for categorical fixed-point operators. In *Proceedings of 15th Annual Symposium on Logic in Computer Science*. IEEE Computer Society, 2000.

[25] P. Wadler. The essence of functional programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages*. ACM, 1992.