Parameterised Notions of Computation

Robert Atkey
LFCS, School of Informatics
University of Edinburgh, Mayfield Road
Edinburgh EH9 3JZ, UK
(*e-mail:* bob.atkey@ed.ac.uk)

Abstract

Moggi's Computational Monads and Power *et al*'s equivalent notion of Freyd category have captured a large range of computational effects present in programming languages. Examples include non-termination, non-determinism, exceptions, continuations, side-effects and input/output. We present generalisations of both computational monads and Freyd categories, which we call *parameterised* monads and *parameterised* Freyd categories, that also capture computational effects with parameters. Examples of such are composable continuations, side-effects where the type of the state varies and input/output where the range of inputs and outputs varies. By also considering structured parameterisation, we extend the range of effects to cover separated side-effects and multiple independent streams of I/O. We also present two typed λ-calculi that soundly and completely model our categorical definitions — with and without symmetric monoidal parameterisation — and act as prototypical languages with parameterised effects.

1 Introduction

Moggi's framework of Computational Monads (Moggi, 1991; Moggi, 1989), and Power *et al*'s equivalent notion of Freyd Categories (Power & Robinson, 1997; Power & Thielecke, 1999; Levy *et al.*, 2003), have been extremely successful in capturing a wide range of computational effects used in programming language designs. Examples include non-termination, non-determinism, exceptions, continuations, side-effects and input/output.

In this paper, we generalise both notions to *parameterised* monads and *parameterised* Freyd categories. The parameterisation will take the form of a parameterising category that will annotate computations with information on their start and end states.

Our motivating example is that of side-effects. The standard side-effects monad selects an object S of some cartesian closed category to represent the type of the computer's store and sets the functor part of the monad to be $TA = (S \times A)^S$. Thus, computations, modelled as the object TA, go from old stores to new stores and values. This monad successfully models global side-effects.

The problem with this solution is that it uses a single object to represent the store at all points during the program. Thus, there is a single "type" that must cover all the possible stores that a program can generate and manipulate. For the purposes of modelling features such as strong update (Morrisett *et al.*, 2005), where the type of storage cells may change over time, or for modelling type systems inspired by Hoare Logic such as Alias Types (Smith *et al.*, 2000), this is inadequate. Such type systems type the current store explicitly and restrict the range of possible operations according to the current store type. In this paper, we propose categorical structure generalising monads to model such situations.

We will present type systems with explicitly typed stores in Sections 3 and 5. An example judgment has the form:

$$\Gamma; S_1 \vdash c : A; S_2$$

The context Γ and type A are the traditional value context and result type respectively. The context S_1 and type S_2 type the initial and final states required and produced by the computation c.

We propose to model this by considering an additional parameterising category $\mathcal{S}$ to interpret the types S_1 and S_2. The arrows of $\mathcal{S}$ are intended to be used to represent effect-free manipulations of store descriptions, analogous to implications between assertions in Hoare Logic. We extend the definition of monad to have underlying functors of type $T : \mathcal{S}^{\mathsf{op}} \times \mathcal{S} \times \mathcal{C} \to \mathcal{C}$, with additional conditions on the unit and multiplication that we set out in Section 2.2. In the case of global state we can assume a functor $\widehat{\cdot} : \mathcal{S} \to \mathcal{C}$ and set $T(S_1, S_2, A) = (\widehat{S_2} \times A)^{\widehat{S_1}}$, or even take $\mathcal{S}$ to be $\mathcal{C}$ itself. Below, we present three other examples, category writers – a generalisation of monoid writers, typed input/output, where the range of inputs and outputs depends on the current type of the state, and Danvy and Filinski's composable continuations (Danvy & Filinski, 1989).

This generalisation of monads suffices for modelling explicitly typed global state. In many cases, however, the assumption that we always know the whole global state is too strong. For example, we can regard the store of a computer as being built from multiple independent regions, right down to the individually addressable storage cells, and a program that only looks at some cells need not concern itself with the rest of the store. Similarly, the computer may have multiple I/O devices attached and be able to send output and receive input from them independently. More abstractly, the state types can denote the possible future effects performed by a program and one small part of the program should not need to know about the whole possible future of the rest of the program.

In order to model this situation, we assume that the parameterising category $\mathcal{S}$ has additional structure. For dealing with separate individual memory cells, the appropriate structure is symmetric monoidal; principally a bifunctor $\otimes : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ that we can use to build composite state descriptions from smaller ones. Hence, if the state types `[Int]` and `[Bool]` represent stores containing an integer and a boolean respectively, the composite state type `[Int]` $\otimes$ `[Bool]` represents a store containing both an integer and a boolean, in separate memory cells.

The problem now is how to sequence two programs operating on separate parts of the heap. If we have arrows $c_1 : A \to T(S_1, S_2, B)$ and $c_2 : B \to T(S_1', S_2', C)$ representing computations, how do we get a single arrow $A \to T(S_1 \otimes S_1', S_2 \otimes S_2', C)$? The solution we present here is to require natural transformations $(- \otimes S)^\dagger$:

$T(S_1, S_2, A) \to T(S_1 \otimes S, S_2 \otimes S, A)$, and symmetrically, to lift computations up to larger state contexts. This lifting is similar to the service provided by monad strength for lifting to larger value contexts, as seen by the definition of *double* parameterised Freyd categories in Section 4, where the two types of computation in context are represented by two premonoidal structures.

Other notions of parameterised monads The definition of parameterised monad we present in this paper is unrelated to Uustalu's *parametrized monads* (Uustalu, 2003). We have chosen the name "Parameterised Monad" for our definition due to the close relationship between our definition and adjunctions with parameters.

Overview In Section 2, we present our definitions of parameterised monad and parameterised Freyd category, prove them equivalent and give our main examples: typed store, typed input/output and composable continuations. We follow this in Section 3 with a typed λ-calculus, the Typed Command Calculus, which is sound and complete for our categorical constructions. In Section 4, we extend our categorical definitions to allow structured parameterisation, extending the range of examples to allow separated store and multiple streams of input/output. We extend the Typed Command Calculus to the situation when the structure is symmetric monoidal in Section 5. Finally, in Sections 6 and 7, we describe related work and present some concluding remarks and future work.

2 Parameterised Notions of Computation

2.1 Computational Monads

We first recall the definition of strong monad and how it is used to model effectful programming languages. We will refer back to this discussion to justify our definitions below.

Moggi (Moggi, 1991) originally proposed the use of strong monads to structure the denotational semantics of programming languages with effects. Instead of explicitly dealing with the semantics of the effect required (exceptions, side-effects, etc.) directly, a semanticist defines a suitable strong monad for their effect in their chosen base category $\mathcal{C}$ (which we assume to have finite products) and builds the rest of the semantics using the monad. A strong monad consists of four parts:

- A functor $T : \mathcal{C} \to \mathcal{C}$;
- A unit $\eta_A : A \to TA$, natural in A;
- A multiplication $\mu_A : TTA \to TA$, natural in A;
- A strength $\tau_{AB} : A \times TB \to T(A \times B)$, natural in A and B.

The basic idea is that an object TA represents computations that yield values of the type represented by the object A. Arrows $A \to TB$ represent programs that yield values of type B with an input of type A. The unit is a computation that sends values to the computation that does nothing but return the value. The multiplication is used to sequence computations. Given computations $f : A \to TB$

and $g : B \to TC$, their sequencing is

$$A \xrightarrow{f} TB \xrightarrow{Tg} TTC \xrightarrow{\mu_C} TC.$$

Intuitively, the multiplication takes a computation that yields a computation that yields a value and sequences them to produce a single computation that yields a value. This definition of sequencing requires the input and output types of programs to match. The strength part of a strong monad is used to rectify this. Given a computation $f : A \to TB$ and an object C representing additional context, we use the strength to get the computation

$$C \times A \xrightarrow{C \times f} C \times TB \xrightarrow{\tau_{CB}} T(C \times B).$$

This computation can now be sequenced with another computation with input of type $C \times B$.

A monad must also obey certain axioms. For unit and multiplication, these state that η is the left and right unit for multiplication ($\eta; \mu = T\eta; \mu = id$) and that multiplication is associative ($T\mu; \mu = \mu; \mu$). These axioms are natural given the computational reading of unit and multiplication. The strength must also obey axioms stating that it commutes with the associativity and unit of $\times$, and the unit and multiplication of the monad.

2.2 Parameterised Strong Monads

In addition to a category $\mathcal{C}$ with finite products, we now also assume an additional category $\mathcal{S}$. The objects of $\mathcal{S}$ represent *state descriptions* and the arrows of $\mathcal{S}$ represent *state manipulations*. A useful intuition here is to think of the objects as assertions about the state, and the arrows as logical entailments.

We introduce the definition of parameterised monad in pieces, describing how it generalises the definition of monad. We extend the underlying functor to be a functor $T : \mathcal{S}^{\mathsf{op}} \times \mathcal{S} \times \mathcal{C} \to \mathcal{C}$. We now consider the object $T(S_1, S_2, A)$ to be a computation that starts in states described by S_1 and ends in states described by S_2, yielding values of type A. On arrows, the functor allows strengthening of pre-state description by contravariance in the first argument and the weakening of post-state description by covariance in the second argument.

The unit of a parameterised monad is a family of arrows $\eta_{SA} : A \to T(S, S, A)$, natural in A and dinatural in S. As with a monad's unit, this unit represents the do-nothing computation, at any state. The dinaturality ((Mac Lane, 1998) §IX.4) requirement amounts to the commutativity of the diagram

$$\begin{tikzcd}
 & T(S,S,A) \arrow[dr, "T(S,f,A)"] & \\
A \arrow[ur, "\eta_{SA}"] \arrow[dr, "\eta_{S'A}"'] & & T(S,S',A) \\
 & T(S',S',A) \arrow[ur, "T(f,S',A)"'] &
\end{tikzcd}$$

for every arrow $f : S_1 \to S_2$ in $\mathcal{S}$. Strengthening of the precondition and weakening of the post-condition are equivalent for the identity computation.

Multiplication of parameterised monads consists of a family of arrows $\mu_{S_1 S_2 S_3 A} : T(S_1, S_2, T(S_2, S_3, A)) \to T(S_1, S_3, A)$. Given computations $f : A \to T(S_1, S_2, B)$ and $g : B \to T(S_2, S_3, C)$, the sequenced computation is

$$A \xrightarrow{f} T(S_1, S_2, B) \xrightarrow{T(S_1,S_2,g)} T(S_1, S_2, T(S_2, S_3, C)) \xrightarrow{\mu_{S_1 S_2 S_3 C}} T(S_1, S_3, C)$$

Hence, only pairs of computations where the former's post-state matches the latter's pre-state may be sequenced. Multiplication is required to be natural in S_1, S_3 and A and dinatural in S_2. Dinaturality in this case amounts to the following diagram commuting for all $f : S_2 \to S_2'$:

$$\begin{tikzcd}
 & T(S_1, S_2', T(S_2', S_3, A)) \arrow[dr, "\mu_{S_1 S_2' S_3 A}"] & \\
T(S_1, S_2, T(S_2', S_3, A)) \arrow[ur, "T(S_1,f,T(S_2',S_3,A))"] \arrow[dr, "T(S_1,S_2,T(f,S_3,A))"'] & & T(S_1, S_3, A) \\
 & T(S_1, S_2, T(S_2, S_3, A)) \arrow[ur, "\mu_{S_1 S_2 S_3 A}"'] &
\end{tikzcd}$$

This states that if we have two computations with a mismatch in the intermediate state that is bridged by $f : S_1 \to S_2$, then it does not matter if we weaken the former's post-state, or the strength latter's pre-state in order to make them match.

The definition of strength generalises easily to parameterised monads. Putting this together, we get:

Definition 1
Given a category $\mathcal{C}$ with finite products and a category $\mathcal{S}$, an *$\mathcal{S}$-parameterised monad* (T, η, μ) on $\mathcal{C}$ consists of:

- A functor $T : \mathcal{S}^{\mathsf{op}} \times \mathcal{S} \times \mathcal{C} \to \mathcal{C}$;
- A unit $\eta_{S,A} : A \to T(S, S, A)$, natural in A and dinatural in S;
- A multiplication $\mu_{S_1,S_2,S_3,A} : T(S_1, S_2, T(S_2, S_3, A)) \to T(S_1, S_3, A)$, natural in S_1, S_3 and A and dinatural in S_2;
- A strength $\tau_{A,S_1,S_2,B} : A \times T(S_1, S_2, B) \to T(S_1, S_2, A \times B)$, natural in A, B, S_1 and S_2.

The unit and multiplication must obey the monad laws: $\eta; \mu = T(S_1, S_2, \eta); \mu = id$ and $T(S_1, S_2, \mu); \mu = \mu; \mu$. The strength must obey the obvious adaptations of the axioms for non-parameterised strength (Moggi, 1991).

An alternative partial definition is given by observing that a non-parameterised monad is equivalent to a one object $\mathcal{C}^{\mathcal{C}}$-enriched category. A multiple object $\mathcal{C}^{\mathcal{C}}$ enriched category is equivalent to part of our definition[1], where the objects are the objects of the parameterising category $\mathcal{S}$. Since a one object normal category

[1] This observation is due to Chung-chieh Shan:
`http://haskell.org/pipermail/haskell-cafe/2004-July/006448.html`

is equivalent to a monoid, we can consider the relationship between parameterised monads and monads as similar to the relationship between monoids and categories. We follow this up below in Section 2.3.3, generalising the monoid writer monad to the category writer parameterised monad. As a special case, if we restrict $\mathcal{S}$ to be the one object, one arrow category then our definition is equivalent to the standard definition of a non-parameterised monad.

2.3 Examples

We now give some examples of parameterised monads modelling computational effects that require and additional parameterising category.

2.3.1 Strong Monads

Every (strong) monad gives a parameterised (strong) monad for *any* parameterising category $\mathcal{S}$. Given a monad (M, η, μ) with optional strength τ, we define an $\mathcal{S}$-parameterised monad (T, η', μ') with optional strength τ' as:

$$\begin{array}{rcl} T(S_1, S_2, A) & = & MA \\ \eta'_{SA} & = & \eta_A \\ \mu'_{S_1 S_2 S_3 A} & = & \mu_A \\ \tau'_{S_1 S_2 AB} & = & \tau_{AB} \end{array}$$

Thus, the resulting parameterised monad just uses the monad, forgetting the parameterisation. The computations available at any pair (S_1, S_2) are always the same.

2.3.2 Typed State

As stated in the introduction, we can use parameterised monads to model typed global state. We assume that our base category $\mathcal{C}$ is cartesian closed, and take $\mathcal{S} = \mathcal{C}$. The parameterised monad's functor is defined as $T(S_1, S_2, A) = (S_2 \times A)^{S_1}$, with the usual unit, multiplication and strength for the global state monad. For each object A of $\mathcal{C}$ we have operations to read and update the current store:

$$\begin{array}{lcllcl} \mathit{read}_A & : & T(A, A, A) & \mathit{store}_{XA} & : & A \to T(X, A, 1) \\ \mathit{read}_A & = & \lambda s.(s, s) & \mathit{store}_{XA} & = & a \mapsto \lambda s.(a, \star) \end{array}$$

By reading the types, we can see that the *read* operation starts in a state where the store is of type A, and ends in a state where the store is of type A, yielding a value of type A – the current value in the store. The *store* operation starts in a state with an arbitrary store type X and replaces it with the supplied value of type A, yielding the trivial element of 1, the terminal object.

Obviously, updating the entire store at once is not very practical; we may wish to consider stores constructed from smaller stores and only read and update individual parts of it at a time. To describe such composite stores we can use $\mathcal{C}$'s cartesian structure, so that a state description $A \times B \times C$ describes a store with three cells, containing an A, a B and a C. We modify the *read* and *store* operations to select parts of the store to operate on:

$$\begin{array}{rcl} \mathit{read}_{S(A)} & : & T(S(A), S(A), A) \\ \mathit{read}_{S(A)} & = & \lambda s.\text{let } S(a) = s \text{ in } (s, a) \\ \\ \mathit{store}_{X,S(A)} & : & A \to T(S(X), S(A), 1) \\ \mathit{store}_{X,S(A)} & = & a \mapsto \lambda s.(S(\mapsto a)s, \star) \end{array}$$

where the notation $S(-)$ denotes a finite product expression $A_1 \times ... \times - \times ... \times A_n$ with a hole. The syntactic sugar "let $S(a) = ...$ in ..." selects the value stored in s at the distinguished location specified by $S(-)$, and $S(\mapsto a)s$ updates the value stored in s at the distinguished location specified by $S(-)$.

Note that on both of the operations we must also record the types of all of the memory cells that *do not* change, as well as the one that does. We rectify this in Section 4 by considering the lifting of the symmetric monoidal structure on states up to the level of computations.

2.3.3 Categories

This example highlights the idea that parameterised monads are to monads as categories are to monoids. For this example, we assume the base category $\mathcal{C} = \text{Set}$.

Recall that, given a monoid $(M, \cdot, e)$, there is a monad T_M on Set, with $T_M(A) = M \times A$, $\eta(a) = (e, a)$ and $\mu(m_1, (m_2, a)) = (m_1 \cdot m_2, a)$. By considering a monoid of traces, with the multiplication as concatenation, this monad can be used to interpret traced computation, or computation with printing.

For the parameterised generalisation, we can consider a small category $\mathcal{S}_1$ instead of a monoid. For the parameterising category we choose some subcategory $\mathcal{S}$ with the same objects as $\mathcal{S}_1$ (a lluf subcategory). We set $T_{\mathcal{S}_1}(S_1, S_2, A) = \mathcal{S}_1(S_1, S_2) \times A$, $\eta_{SA}(a) = (id_S, a)$ and $\mu_{S_1S_2S_3A}((s_1, (s_2, a))) = (s_1; s_2, a)$.

As an application of this construction, consider the category StkPrg of simple stack machine programs, which is the free category on the graph whose objects are natural numbers denoting stack depths and edges are lists of commands freely generated by the rules:

$$\frac{}{n \xrightarrow{[]} n} \qquad \frac{i \in \mathbb{Z}}{n \xrightarrow{[\mathsf{push}.i]} n+1} \qquad \frac{}{n+2 \xrightarrow{[\mathsf{add}]} n+1} \qquad \frac{}{n+1 \xrightarrow{[\mathsf{dup}]} n+2}$$

$$\frac{n_1 \xrightarrow{[\overrightarrow{c_1}]} n_2 \qquad n_2 \xrightarrow{[\overrightarrow{c_2}]} n_3}{n_1 \xrightarrow{[\overrightarrow{c_1}, \overrightarrow{c_2}]} n_3}$$

Composition of $[\overrightarrow{c_1}] : n_1 \to n_2$ and $[\overrightarrow{c_2}] : n_2 \to n_3$ is defined as $[\overrightarrow{c_1}, \overrightarrow{c_2}]$, which is an arrow by these rules. Identities are given by the empty list, which is an arrow at every numeral by the first rule. Note that this category is just a special case of programs with specified start and end specifications.

Taking T_{StkPrg} as defined above, with $|\mathrm{StkPrg}|$, the discrete category with the same objects as StkPrg as the parameterising category, we can define the following basic operations for the monad:

$$\begin{array}{rcl} push_n : \mathbb{Z} \to T_{\mathrm{StkPrg}}(n, n+1, 1) & = & i \mapsto ([\mathsf{push}.i], \star) \\ add_n : 1 \to T_{\mathrm{StkPrg}}(n+2, n+1, 1) & = & \star \mapsto ([\mathsf{add}], \star) \\ dup_n : 1 \to T_{\mathrm{StkPrg}}(n+1, n+2, 1) & = & \star \mapsto ([\mathsf{dup}], \star) \end{array}$$

where $\mathbb{Z}$ is the usual set of integers.

Thus, computations in T_{StkPrg} model programs that construct stack machine programs that do not have the possibility of stack under-flow at run-time. One could also envisage more complex examples involving typed stacks and jumps to labels, and even the construction of programs satisfying Hoare logic specifications. Using typed stacks with subtyping relationships between the types of elements on the stack would extend this example to non-discrete parameterising categories: arrows of $\mathcal{S}$ would model the subtyping relations between stacks.

Note that we have had to index the operations by the height of the stack at the current point in the abstract machine. We will rectify this by lifting the addition operation on the objects of StkPrg up to computations themselves in Section 4.

2.3.4 Typed I/O

The monad of the previous example essentially models constrained output; the range of possible outputs at each stage of the program is determined by the outputs that have gone before. In this example, we generalise to also allow inputs, where the types of the possible values input, as well as the possible outputs, are dependent on the current state.

Again, for simplicity, we restrict to $\mathcal{C} = \mathrm{Set}$. Take $\mathcal{S}$ to be a small category. The objects of $\mathcal{S}$ will represent the states of an input/output device, while arrows $S_1 \to S_2$ will be witnesses for proofs that S_1 allows all the operations that S_2 allows.

Let Ω be a set of I/O operations. For each $op \in \Omega$, we assume there are two sets:

$\mathsf{in}(op)$ **:** The set of values that can be input by performing the operation op;

$\mathsf{out}(op)$ **:** The set of values that can be output by performing the operation op.

We further assume that every operation op has two associated objects of $\mathcal{S}$:

$\mathsf{pre}(op)$ **:** The state in which the operation op may be performed;

$\mathsf{post}(op)$ **:** The state that results after the operation op has been performed.

Given such a collection of operations Ω, we construct a monad T_Ω. On objects –

i.e. sets – the functor part is built by these inductive rules:

$$\frac{a \in A \qquad f : S \to S'}{\mathsf{e}(f, a) \in T_\Omega(S, S', A)}$$

$$\frac{op \in \Omega \qquad o \in \mathsf{out}(op) \qquad k \in \mathsf{in}(op) \to T_\Omega(\mathsf{post}(op), S', A) \qquad f : S \to \mathsf{pre}(op)}{\mathsf{o}(f, op, o, k) \in T_\Omega(S, S', A)}$$

Computations in T_Ω are therefore trees with values at the leaves and operations at the nodes, branching on the possible input values for each operation. Between each node there is an arrow of $\mathcal{S}$, acting as a witness that the operations are compatible.

On arrows of $\mathcal{S}$, $T_\Omega(f, g, A)$ pre-composes f to the $\mathcal{S}$-arrow at the root of the tree and post-composes g to all the $\mathcal{S}$-arrows at the leaves of the tree. On functions $f : A \to B$, $T_\Omega(S_1, S_2, f)$ performs the usual "map" operation on trees. The monad unit maps a to e(id, a) and multiplication concatenates trees, pre-composing the final $\mathcal{S}$-arrow in each leaf of the first tree with the root of the second tree.

For each operation $op \in \Omega$, there is a primitive operation of the monad:

$$\begin{array}{lcl} perform_{op} & : & \mathsf{out}(op) \to T_\Omega(\mathsf{pre}(op), \mathsf{post}(op), \mathsf{in}(op)) \\ perform_{op} & = & o \mapsto \mathsf{o}(id, op, o, \lambda i.\mathsf{e}(i)) \end{array}$$

Note the apparent swapping of the meanings of in and out as input and output from the point of view of operations on the monad.

We give two examples of this construction.

A Stateful I/O Device We assume some device with three states inactive, initialising and active. There are six operations, shown in Table 1. The idea of this example is that the I/O device initially starts in the state inactive. The operation "activate" moves the device into the state initialising, where the client can issue initialisation data – here represented as booleans – to the device via the operation "initData". On the operation "finishInit", the device moves to the state active, where the client can use the "read" and "write" operations to read and write data – here represented by integers – from the device. Finally, the client issues "shutdown" to reset the device back to inactive, returning a status code as it does so. In this case the category $\mathcal{S}$ consists of an object for each of the states, and no arrows.

Session Types Our second example of Typed I/O involves a simple form of session types (Vasconcelos *et al.*, 2006) (see also the similar concepts of history effects (Skalka & Smith, 2004) and behaviour effects (Nielson & Nielson, 1996)). Let X, X_1, X_2, etc. be a collection of sets of values suitable for input/output. The states descriptions in this case are abstract traces of possible I/O behaviour that a program may take – i.e. sessions – given by the grammar

$$S \quad ::= \quad ?X \ \mid\ !X \ \mid\ S_1 + S_2 \ \mid\ S_1.S_2 \ \mid\ \circ.$$

A session $?X$ indicates the that program must input a value in X and $!X$ indicates that the program must output a value in X. The combination $S_1 + S_2$ means that the program has the choice of doing either the actions in S_1 or the actions in

op	$\mathsf{pre}(op)$	$\mathsf{post}(op)$	$\mathsf{out}(op)$	$\mathsf{in}(op)$
activate	inactive	initialising	1	1
initData	initialising	initialising	$\mathbb{B}^a$	1
finishInit	initialising	active	1	1
read	active	active	1	$\mathbb{Z}$
write	active	active	$\mathbb{Z}$	1
shutdown	active	inactive	1	$\mathbb{Z}$

[a] $\mathbb{B}$ is the set $\{\text{true}, \text{false}\}$ of boolean values.

Table 1. *Stateful I/O device operations*

op	$\mathsf{pre}(op)$	$\mathsf{post}(op)$	$\mathsf{out}(op)$	$\mathsf{in}(op)$
$\text{input}_{X,S}$	$?X.S$	S	1	X
$\text{output}_{X,S}$	$!X.S$	S	X	1

Table 2. *Session types I/O operations*

S_2, whereas the combination $S_1.S_2$ prescribes that the program must perform the operations in S_1 and then the operations in S_2. The session $\circ$ indicates that no action is possible. The arrows of $\mathcal{S}$ are those given by the smallest preorder that treats $S_1.S_2$ as an associative binary operations with unit $\circ$ and $S_1 + S_2$ as a meet.

The operations are shown in Table 2. Note that there are infinitely many operations indexed by the input/output value sets X as well as all the possible future sessions S. The primitive operations on the monad have the types:

$$\begin{array}{rcl} \text{input}_{X,S} & : & 1 \to T(?X.S, S, X) \\ \text{output}_{X,S} & : & X \to T(!X.S, S, 1). \end{array}$$

As with the monads T_{GS_2} and T_{StkPrg} above, we have the problem that the primitive operations at the monad level have to explicitly declare all of the following session. This problem becomes especially apparent when we attempt to use the operations in the Typed Command Calculus defined in Section 3. We will rectify this in Section 4 by lifting the structure of the sessions up to computations.

2.3.5 Composable Continuations

Parameterised monads provide a way to interpret Danvy and Filinski's composable continuations (Danvy & Filinski, 1989). Composable continuations provide access to evaluation contexts smaller than the whole program, delimited at runtime by the *reset* operator. The current context is made available to the program by the *shift* operator. In contrast, the *call with current continuation* operator only allows the entire program to be treated as the current context. The following is inspired by Wadler's expression of composable continuations in terms of monads (Wadler, 1994).

We require $\mathcal{C}$ to be cartesian closed, and set $\mathcal{S}$ to be $\mathcal{C}^{\mathsf{op}}$. Define $T(R_1, R_2, A) = (A \to R_2) \to R_1$, where $\to$ is the exponential functor. Unit, multiplication and strength are defined as for the standard continuations monad (Moggi, 1991). We write the definitions out using $\mathcal{C}$'s internal language:

$$\begin{aligned} \eta(x) &= \lambda k.kx \\ \mu(f) &= \lambda k.f(\lambda k'.k'k) \\ \tau(a, f) &= \lambda k.f(\lambda b.k(a, b)) \end{aligned}$$

In terms of the type system given by Danvy and Filinski in (Danvy & Filinski, 1989), a judgment $\rho, \alpha \vdash E : \tau, \beta$ is interpreted as an arrow $[\![\rho]\!] \to T([\![\beta]\!], [\![\alpha]\!], [\![\tau]\!])$. The *reset* operator is interpreted as an arrow in $\mathcal{C}$, using $\mathcal{C}$'s internal language:

$$\begin{aligned} \mathit{reset} &: T(B, A, A) \to T(C, C, B) \\ \mathit{reset} &= c \mapsto \lambda k.k(c(\lambda x.x)) \end{aligned}$$

Thus *reset* calls its argument c with the empty continuation, represented by the identity function, and feeds the output to the current continuation. The shift operator is defined as:

$$\begin{aligned} \mathit{shift} &: ((A \to T(C, C, B)) \to T(E, D, D)) \to T(E, B, A) \\ \mathit{shift} &= f \mapsto \lambda k.f(\lambda v.\eta(kv))(\lambda x.x) \end{aligned}$$

Applied to f, *shift* calls f with a function that, given an A, invokes the current surrounding context (up to the closest dynamically enclosing *reset*) and returns the answer. The resulting computation is then invoked with the empty continuation. The shift operator therefore takes the current continuation and makes it available to the program. The extra type information is essential here due to the fact that continuation contexts need not extend to the whole program, so the result types in the continuation depend on the rest of the program.

Due to the fact that our "state" category in this example is the opposite of our base category, we can use to the functorial action of the monad on its first two parameters to get an operation, which we call side:

$$\begin{aligned} \mathit{side} &: \mathcal{C}(A, B) \to \mathcal{C}(1, T(B, A, 1)) \\ \mathit{side}(f) &= \lambda k.f(k\star) \end{aligned}$$

where 1 is the terminal object in $\mathcal{C}$ and $\star$ is its unique value in the internal language. Another way of expressing this is as $\eta_{B1}; T(f, B, 1)$ which is equal to $\eta_{A1}; T(A, f, 1)$ by dinaturality. The effect of this operation is to postcompose the current continuation with the argument f, meaning that f will be run on the result after the rest of the computation in the current context.

Although we have derived this operation from the functorial action of T on its state parameters, which was not available to Danvy and Filinski, we have not increased the expressive power of the type system. The new operation is expressible in terms of *shift*. If we define *side'* as:

$$side'(f) \quad = \quad shift(\lambda c.bind(c(\star), \lambda x.\eta(fx)))$$

where $bind : T(S_1, S_2, A) \times (A \to T(S_2, S_3, B)) \to T(S_1, S_3, B)$ is the monadic bind operator derived from μ. Hence *side'* uses *shift* to obtain the current context, runs it, and applies f to the result before returning. The two operations *side* and *side'* are easily seen to be equivalent by unwinding all the definitions and rewriting using the $\beta\eta$ rules.

in Section 3.2.1 below we give some examples of the use of composable continuations in our typed calculus. See also (Danvy & Filinski, 1989) and (Wadler, 1994) for examples of the use of *shift* and *reset*. This example needs much more work to establish the precise categorical properties of *shift* and *reset*, and to potentially axiomatise it without reference to an underlying continuation passing interpretation, following the lead set by Thielecke (Thielecke, 1997).

2.3.6 Change of State Category

Finally in this sequence of examples, we note that if we are given any functor $F : \mathcal{S}' \to \mathcal{S}$ and an $\mathcal{S}$-parameterised monad (T, η, μ) then we can define an $\mathcal{S}'$-parameterised monad by:

$$\begin{aligned} T'(S'_1, S'_2, A) &= T(FS'_1, FS'_2, A) \\ \eta'_{S'A} &= \eta_{F(S')A} \\ \mu'_{S'_1S'_2S'_3A} &= \mu_{F(S'_1)F(S'_2)F(S'_3)A} \end{aligned}$$

If T also has a strength τ, then we can define $\tau'_{AS'_1S'_2B} = \tau_{AF(S'_1)F(S'_2)B}$.

2.4 Parameterised Freyd Categories

Freyd categories are comprised of identity on objects functors $J : \mathcal{C} \to \mathcal{K}$, where $\mathcal{K}$ has premonoidal structure and J strictly preserves it by seeing the finite product structure of $\mathcal{C}$ as premonoidal structure. Premonoidal structure consists of a family of pairs of functors $A \otimes - : \mathcal{K} \to \mathcal{K}$ and $- \otimes A : \mathcal{K} \to \mathcal{K}$ that agree on objects: $A \otimes B = A \otimes B = A \otimes B$, and associativity, left and right unit and symmetry natural transformations as for symmetric monoidal structure. The components of these natural transformations must be *central*: an arrow f of $\mathcal{K}$ is central if, for all arrows g of $\mathcal{K}$, we have $A \otimes f; g \otimes B' = g \otimes B; A' \otimes f$, i.e. f commutes with g when they operate on different values. Arrows of $\mathcal{K}$ are used to represent computations, with the identity arrow representing the identity computation, and composition representing the sequencing of computations. The premonoidal structure is used to represent computation in context.

Our definition of parameterised Freyd category builds the required structure in a single step, unlike the two steps of premonoidal structure on the codomain category, and then a strict premonoidal functor as for Freyd categories. We do it in this way for two reasons. Firstly, we want the objects of the codomain category to be comprised of pairs of objects of the value and state categories but with the premonoidal structure only referring to the value category, so we start by requiring an identity on objects functor $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$. The premonoidal structure is then built on top of this, building in the requirement of strict preservation of premonoidal structure. Secondly, there is no directly analogous definition of centrality for arrows in a parameterised Freyd category, due to the composition ordering imposed by the objects of the state category. Therefore we just state that the symmetric monoidal structure natural transformations of $\mathcal{C}$ via J are the ones we need, rather than requiring them on $\mathcal{K}$ with J preserving them.

Definition 2
A *parameterised Freyd category* consists of three categories $\mathcal{C}$, $\mathcal{S}$ and $\mathcal{K}$, where $\mathcal{C}$ has finite products, and three functors $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$, $\olessthan_{\mathcal{C}} : \mathcal{C} \times \mathcal{K} \to \mathcal{K}$ and $\ogreaterthan_{\mathcal{C}} : \mathcal{K} \times \mathcal{C} \to \mathcal{K}$, such that:

1. J is identity on objects;
2. The monoidal structure of $\mathcal{C}$ is respected: $A \olessthan_{\mathcal{C}} J(B, X) = J(A, X) \ogreaterthan_{\mathcal{C}} B = J(A \times B, X)$ and $f \olessthan_{\mathcal{C}} J(g, s) = J(f, s) \ogreaterthan_{\mathcal{C}} g = J(f \times g, s)$;
3. For each $S \in \mathsf{Ob}\mathcal{S}$, the transformations given by the associativity $J(\alpha_{ABC}, S)$, the left unit $J(\lambda_A, S)$, the right unit $J(\rho_A, S)$ and the symmetry $J(\sigma_{A,B}, S)$ of the symmetric monoidal structure arising from $\mathcal{C}$'s finite products must be natural in the variables in all combinations of $\times, \olessthan_{\mathcal{C}}$ and $\ogreaterthan_{\mathcal{C}}$ that make up their domain and codomain. For example, for associativity, the following diagrams must commute:

$$\begin{tikzcd}
A \olessthan_{\mathcal{C}} (B \olessthan_{\mathcal{C}} (C, S)) \arrow[r, "J(\alpha, S)"] \arrow[d, "f \olessthan_{\mathcal{C}} (g \olessthan_{\mathcal{C}} c)"'] & (A \times B) \olessthan_{\mathcal{C}} (C, S) \arrow[d, "(f \times g) \olessthan_{\mathcal{C}} c"] \\
A' \olessthan_{\mathcal{C}} (B' \olessthan_{\mathcal{C}} (C', S')) \arrow[r, "J(\alpha, S)"] & (A' \times B') \olessthan_{\mathcal{C}} (C', S')
\end{tikzcd}$$

$$\begin{tikzcd}
A \olessthan_{\mathcal{C}} ((B, S) \ogreaterthan_{\mathcal{C}} C) \arrow[r, "J(\alpha, S)"] \arrow[d, "f \olessthan_{\mathcal{C}} (c \ogreaterthan_{\mathcal{C}} g)"'] & (A \olessthan_{\mathcal{C}} (B, S)) \ogreaterthan_{\mathcal{C}} C \arrow[d, "(f \olessthan_{\mathcal{C}} c) \ogreaterthan_{\mathcal{C}} g"] \\
A' \olessthan_{\mathcal{C}} ((B', S') \ogreaterthan_{\mathcal{C}} C') \arrow[r, "J(\alpha, S)"] & (A' \olessthan_{\mathcal{C}} (B, S')) \ogreaterthan_{\mathcal{C}} C'
\end{tikzcd}$$

$$\begin{tikzcd}
(A, S) \ogreaterthan_{\mathcal{C}} (B \times C) \arrow[r, "J(\alpha, S)"] \arrow[d, "c \ogreaterthan_{\mathcal{C}} (f \times g)"'] & ((A, S) \ogreaterthan_{\mathcal{C}} B) \ogreaterthan_{\mathcal{C}} C \arrow[d, "(c \ogreaterthan_{\mathcal{C}} f) \ogreaterthan_{\mathcal{C}} g"] \\
(A', S') \ogreaterthan_{\mathcal{C}} (B' \times C') \arrow[r, "J(\alpha, S)"] & ((A', S') \ogreaterthan_{\mathcal{C}} B') \ogreaterthan_{\mathcal{C}} C'
\end{tikzcd}$$

and similarly for left and right unit and symmetry.

This definition can be split into two parts: the functor $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$, which identifies how pure value computations and state manipulations are incorporated into commands; and the premonoidal structure with respect to $\mathcal{C}$, given by the functors $\olessthan_{\mathcal{C}}$ and $\ogreaterthan_{\mathcal{C}}$. Closure for parameterised Freyd categories is similar to that for Freyd categories. It will be used to interpret function types.

Definition 3
A parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ is *closed* if, for all $A \in \mathsf{Ob}\mathcal{C}$ and $S \in \mathsf{Ob}\mathcal{S}$, the functor $J(- \times A, S) : \mathcal{C} \to \mathcal{K}$ has a specified right adjoint, written $(A, S) \to - : \mathcal{K} \to \mathcal{C}$.

2.4.1 Parameterised Monads and Parameterised Freyd Categories

We now show the relationship between parameterised Freyd categories and strong parameterised monads. To do this we shall go through parameterised adjunctions. We will show that parameterised monads have the same relationship with parameterised adjunctions as monads have with adjunctions. Since a closed Freyd category is a parameterised adjunction, this will give a way of constructing parameterised monads from parameterised Freyd categories. In the opposite direction, there is a natural definition of Kleisli category for a parameterised monad. When the parameterised monad is strong, this will give a parameterised Freyd category.

Definition 4
An *$\mathcal{S}$-parameterised adjunction* from $\mathcal{C}$ to $\mathcal{D}$ is a 4-tuple $\langle F, G, \eta, \epsilon \rangle : \mathcal{C} \to \mathcal{D}$ where F and G are functors:

$$F : \mathcal{S} \times \mathcal{C} \to \mathcal{D} \qquad\qquad G : \mathcal{S}^{\mathsf{op}} \times \mathcal{D} \to \mathcal{C}$$

and η and ϵ are the unit and counit, natural in A and dinatural in S:

$$\eta_{SA} : A \to G(S, F(S, A)) \qquad\qquad \epsilon_{SA} : F(S, G(S, A)) \to A$$

subject to the triangle identities:

$$\begin{tikzcd}
G(S,A) \arrow[r, "\eta_{SG(S,A)}"] \arrow[dr, "id"'] & G(S, F(S, G(S, A))) \arrow[d, "G(S,\epsilon_{SA})"] \\
& G(S,A)
\end{tikzcd}
\qquad
\begin{tikzcd}
F(S,A) \arrow[r, "F(S,\eta_{SA})"] \arrow[dr, "id"'] & F(S, G(S, F(S, A))) \arrow[d, "\epsilon_{SF(S,A)}"] \\
& F(S,A)
\end{tikzcd}$$

By Theorem §IV.7.3 in (Mac Lane, 1998), if we have a functor $F : \mathcal{S} \times \mathcal{C} \to \mathcal{D}$ such that for every object S, $F(S, -)$ has a right adjoint $G_S : \mathcal{D} \to \mathcal{C}$, then there is a unique way to make G into a bifunctor $\mathcal{S}^{\mathsf{op}} \times \mathcal{D} \to \mathcal{C}$ such that the pair form a parameterised adjunction in the sense of this definition. Using this, we can turn the closed structure of a closed parameterised Freyd category into an $\mathcal{S}$-parameterised adjunction between $\mathcal{C}$ and $\mathcal{K}$ with the functors $J(-, S)$ and $(1, S) \to -$.

Parameterised monads are to parameterised adjunctions as monads are to adjunctions, as the following lemma partially demonstrates. It also possible to define a suitable notion of Eilenberg-Moore category of algebras for a parameterised monad, and this and the Kleisli category used in this lemma are the final and initial objects in the category of adjunctions defining the parameterised monad, as for monads. See the appendix of (Atkey, 2006) for more details.

Proposition 1
$\mathcal{S}$-parameterised adjunctions $\langle F, G, \eta, \epsilon \rangle : \mathcal{C} \to \mathcal{D}$ give $\mathcal{S}$-parameterised monads on $\mathcal{C}$, defined as:

$$T(S_1, S_2, A) = G(S_1, F(S_2, A)) \qquad \eta^T_{S,A} = \eta_{S,A} \qquad \mu^T_{S_1,S_2,S_3,A} = G(S_1, \epsilon_{S_2,F(S_3,A)})$$

Conversely, given an $\mathcal{S}$-parameterised monad on $\mathcal{C}$, if we define a category $\mathcal{C}_T$ with objects pairs of $\mathcal{C}$ and $\mathcal{S}$ objects; and homsets:

$$\mathcal{C}_T((A_1, S_1), (A_2, S_2)) = \mathcal{C}(A_1, T(S_1, S_2, A_2)).$$

then the functors

$$\begin{array}{rcl} F & : & \mathcal{S} \times \mathcal{C} \to \mathcal{C}_T \\ F(S, A) & = & (A, S) \\ F(s, f) & = & \eta; T(S_1, s, f) \end{array}$$

and

$$\begin{array}{rcl} G & : & \mathcal{S}^{\mathsf{op}} \times \mathcal{C}_T \to \mathcal{C} \\ G(S_1, (A, S_2)) & = & T(S_1, S_2, A) \\ G(s, c) & = & T(s, S_2, c); \mu \end{array}$$

form a parameterised adjoint pair.

Proof
Almost identical to the proof of the definition of a monad from an adjunction (Mac Lane, 1998). The additional (di)naturality conditions are easy to check. The second part is just the parameterised generalisation of the construction of the Kleisli category. □

Thus, every closed Freyd category gives a parameterised monad, and we can generate a category $\mathcal{K}$ and an identity on objects functor $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ via the parameterised version of the Kleisli construction. We extend Power and Robinson's Theorem 4.2 of (Power & Robinson, 1997), which links the premonoidal structure of Freyd categories with monad strength, to the parameterised case:

Proposition 2
Given an strength for a parameterised monad (T, η, μ), there is premonoidal structure on $\mathcal{C}_T$ with respect to $\mathcal{C}$, and *vice versa*. These constructions are inverse.

Proof
Given a strength τ, define $f \otimes_{\mathcal{C}} c = f \times c; \tau_{A,S_1,S_2,B}$, $\otimes_{\mathcal{C}}$ is similar. Given premonoidal structure $\otimes_{\mathcal{C}}$, define $\tau_{A,S_1,S_2,B} = id_A \otimes_{\mathcal{C}} id_{T(S_1,S_2,B)}$ as an arrow of $\mathcal{C}$, where $id_{T(S_1,S_2,B)}$ is seen as an arrow $T(S_1, S_2, B) \to B$ in $\mathcal{C}_T$. The axioms in each case are easily checked. That these operations are inverse is seen by writing out the two definitions and calculating, keeping careful track of the different compositions in $\mathcal{C}$ and $\mathcal{C}_T$. □

Proposition 3
If a strong parameterised monad has Kleisli exponentials, i.e. there is a functor $(B, S_1) \to - : \mathcal{C}_T \to \mathcal{C}$ for all objects B, S_1, and a natural isomorphism $\mathcal{C}_T((A \times B, S_1), (C, S_2)) \cong \mathcal{C}(A, (B, S_1) \to (C, S_2))$, then the induced parameterised Freyd

category is closed. Conversely, every closed parameterised Freyd category gives a strong monad with Kleisli exponentials. These operations are inverse.

Proof
The closed structure is identical in both cases. □

These propositions combine to give:

Theorem 1
Strong parameterised monads with Kleisli exponentials and closed parameterised Freyd categories are equivalent.

3 Typed Command Calculus

We now define a typed λ-calculus, which we call the Typed Command Calculus, which is sound and complete for parameterised Freyd categories. The design of the calculus is based on the fine-grain call-by-value calculus for Freyd categories given by Levy, Power and Thielecke (Levy *et al.*, 2003).

Levy *et al*'s fine-grain call-by-value calculus differs from Moggi's λ_{c} calculus (Moggi, 1989) and Moggi's monadic metalanguage (Moggi, 1991) by making a syntactic distinction between *producers*, which may perform effects in the monad, and *values*, which perform no effects. In this terminology, the λ_{c} calculus treats all terms as producers, and the monadic metalanguage treats all terms as values. The syntactic distinction clarifies the presentation of the calculus, and is based on the structure of Freyd categories.

The fine-grain call-by-value calculus has two typing judgments $\Gamma \vdash^{\mathsf{v}} V : A$ and $\Gamma \vdash^{\mathsf{p}} M : A$. The first is used to type values, and the second is used to type producers. The two main constructs of the calculus are typed as follows:

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A}{\Gamma \vdash^{\mathsf{p}} \texttt{produce}\ V : A} \qquad \frac{\Gamma \vdash^{\mathsf{p}} M : A \qquad \Gamma, x : A \vdash^{\mathsf{p}} N : B}{\Gamma \vdash^{\mathsf{p}} M\ \texttt{to}\ x.N : B}$$

The construct $\texttt{produce}\ V$ incorporates values into producers by treating them as a computation with no effect that returns the given value. The construct $M\ \texttt{to}\ x.N$ denotes the execution of the effectful computation M in the context Γ, feeding its result to N which is then executed.

The fine-grain call-by-value calculus is interpreted in a Freyd category $J : \mathcal{C} \to \mathcal{K}$. Judgments of the value component are interpreted in $\mathcal{C}$ and judgments of the computation component are interpreted in the category $\mathcal{K}$. The $\texttt{produce}\ V$ construct is interpreted using the functor J, and the sequencing construct $M\ \texttt{to}\ x.N$ is interpreted using the premonoidal structure and composition.

3.1 Typing Rules

We follow the basic structure of the fine-grain call-by-value calculus, though we superficially alter the syntax to fit better with the syntax of Moggi's computational λ-calculus (Moggi, 1989). The Typed Command Calculus has a typing judgment for each category present in the definition of parameterised Freyd category. For the three categories, there are three judgments:

$$S_1 \vdash^{\mathsf{s}} s : S_2 \qquad\qquad \Gamma \vdash^{\mathsf{v}} e : A \qquad\qquad \Gamma; S_1 \vdash^{\mathsf{c}} c : A; S_2$$

The first is used to type state manipulations, and is interpreted in the state category $\mathcal{S}$. State manipulation terms are lists of primitive state manipulations. The second is used to type values, and will be interpreted in the value category $\mathcal{C}$. Value terms are comprised of variables, units, pairs, projections, primitive functions and function abstractions, but not applications. The third judgment form is used to type computations, and is interpreted in the category $\mathcal{K}$. Computation terms are comprised of combinations of pure value and state terms, sequencing, primitive computations and function application. Formally, the terms for each of the three calculi are given by the grammar

$$\begin{array}{lcl} s & ::= & \cdot \mid s.\mathtt{m} \\ e & ::= & x \mid \mathtt{f}\ e \mid \star_1 \mid (e_1, e_2) \mid \pi_i e \mid \lambda(x^A; S).c \\ c & ::= & (e; s) \mid \mathrm{let}\ x \Leftarrow c_1\ \mathrm{in}\ c_2 \mid \mathtt{p}\ e \mid e_1\ e_2 \end{array}$$

where $\mathtt{m}$, $\mathtt{f}$ and $\mathtt{p}$ range over given sets of typed primitive state manipulations Φ_S, value functions Φ_V and computations Φ_C respectively. The state types $S, S_1, S_2, ...$ are assumed as given. Value types are generated by the grammar

$$A \quad ::= \quad X \in \mathcal{T}_V \mid 1 \mid A_1 \times A_2 \mid (A_1; S_1) \to (A_2; S_2)$$

where X ranges over a set $\mathcal{T}_V$ of primitive value types. Value type contexts Γ consist of lists of variable name-type pairs with no duplicate names. The typing rules are shown in Figure 1.

The state calculus is very simple, reflecting the fact that we have not required any structure on the state category $\mathcal{S}$ in our models. The rule S-Id types the identity state manipulation that does nothing. The rule S-Prim types the use of primitive state manipulations.

The value and command calculi are mutually defined via the rules for function abstraction and application, V-$\to$I and C-$\to$E. The value calculus includes the standard rules for variables, products and the unit type. The rule V-Prim types the use of primitive functions. The rule V-$\to$I introduces functions. Since a function represents a suspended computation, it is treated as a pure value; this rule takes a judgment in the command calculus and produces one in the value calculus. The rule C-$\to$E eliminates functions, producing an effectful computation in the command calculus. Functions are to be interpreted using the closed structure of a closed parameterised Freyd category.

The rule C-V-S incorporates the terms of the value and state calculi into the command calculus. This rule will be interpreted by the action of the functor J. The C-Prim rule types primitive commands. The rule C-Let sequences two computations similar to the M `to` $x.N$ construct of the fine-grain call-by-value calculus.

Substitution of value expressions e into other value expressions and computations

State Calculus:

$$\frac{}{S \vdash^{\mathsf{s}} \cdot : S}\ (\text{S-Id}) \qquad \frac{S_1 \vdash^{\mathsf{s}} s : S_2 \qquad (\mathtt{m} : S_2 \longrightarrow S_3) \in \Phi_S}{S_1 \vdash^{\mathsf{s}} s.\mathtt{m} : S_3}\ (\text{S-Prim})$$

Value Calculus:

$$\frac{x : A \in \Gamma}{\Gamma \vdash^{\mathsf{v}} x : A}\ (\text{V-Var}) \qquad \frac{\Gamma \vdash^{\mathsf{v}} e : A_1 \qquad (\mathtt{f} : A_1 \longrightarrow A_2) \in \Phi_V}{\Gamma \vdash^{\mathsf{v}} \mathtt{f}\ e : A_2}\ (\text{V-Prim})$$

$$\frac{}{\Gamma \vdash^{\mathsf{v}} \star_1 : 1}\ (\text{V-1I}) \qquad \frac{\Gamma \vdash^{\mathsf{v}} e_1 : A_1 \qquad \Gamma \vdash^{\mathsf{v}} e_2 : A_2}{\Gamma \vdash^{\mathsf{v}} (e_1, e_2) : A_1 \times A_2}\ (\text{V-}\times\text{I})$$

$$\frac{\Gamma \vdash^{\mathsf{v}} e : A_1 \times A_2}{\Gamma \vdash^{\mathsf{v}} \pi_i e : A_i}\ (\text{V-}\times\text{E-}i) \qquad \frac{\Gamma, x : A_1; S_1 \vdash^{\mathsf{c}} c : A_2; S_2}{\Gamma \vdash^{\mathsf{v}} \lambda(x^{A_1}; S_1).c : (A_1; S_1) \to (A_2; S_2)}\ (\text{V-}\to\text{I})$$

Command Calculus:

$$\frac{\Gamma \vdash^{\mathsf{v}} e : A \qquad S_1 \vdash^{\mathsf{s}} s : S_2}{\Gamma; S_1 \vdash^{\mathsf{c}} (e; s) : A; S_2}\ (\text{C-V-S})$$

$$\frac{\Gamma \vdash^{\mathsf{v}} e : A \qquad (\mathtt{p} : (A; S_1) \longrightarrow (B; S_2)) \in \Phi_C}{\Gamma; S_1 \vdash^{\mathsf{c}} \mathtt{p}\ e : B; S_2}\ (\text{C-Prim})$$

$$\frac{\Gamma; S_1 \vdash^{\mathsf{c}} c_1 : A; S_2 \qquad \Gamma, x : A; S_2 \vdash^{\mathsf{c}} c_2 : B; S_3}{\Gamma; S_1 \vdash^{\mathsf{c}} \text{let } x \Leftarrow c_1 \text{ in } c_2 : B; S_3}\ (\text{C-Let})$$

$$\frac{\Gamma \vdash^{\mathsf{v}} e_1 : (A; S_1) \to (B; S_2) \qquad \Gamma \vdash^{\mathsf{v}} e_2 : A}{\Gamma; S_1 \vdash^{\mathsf{c}} e_1\ e_2 : B; S_2}\ (\text{C-}\to\text{E})$$

Fig. 1. Typing rules for the Typed Command Calculus

is standard:

$$\begin{array}{rcll} y[e/x] & = & \begin{cases} x & \text{if } x \neq y \\ e & \text{if } x = y \end{cases} & \\ (\mathtt{f}\ e')[e/x] & = & \mathtt{f}\ (e'[e/x]) & \\ \star[e/x] & = & \star & \\ (e_1, e_2)[e/x] & = & (e_1[e/x], e_2[e/x]) & \\ (\pi_i e')[e/x] & = & \pi_i(e'[e/x]) & \\ (\lambda(y^A; S).c)[e/x] & = & \lambda(y^A; S).(c[e/x]) & \quad y \text{ fresh for } e \end{array}$$

and

$$\begin{array}{rcll} (e'; s)[e/x] & = & (e'[e/x]; s) & \\ (\mathtt{p}\ e')[e/x] & = & \mathtt{p}\ (e'[e/x]) & \\ (\text{let } y \Leftarrow c_1 \text{ in } c_2)[e/x] & = & \text{let } y \Leftarrow c_1[e/x] \text{ in } c_2[e/x] & \quad y \text{ fresh for } e \\ (e_1'\ e_2')[e/x] & = & (e_1'[e/x], e_2'[e/x]) & \end{array}$$

The substitution of state manipulations into computations is really just a prepending operation:

$$\begin{array}{rcl} (e; s')[s/\cdot] & = & (e; s.s') \\ (\mathtt{p}\ e)[s/\cdot] & = & \mathtt{p}\ e \\ (\text{let } x \Leftarrow c_1 \text{ in } c_2)[s/\cdot] & = & \text{let } x \Leftarrow c_1[s/\cdot] \text{ in } c_2 \\ (e_1\ e_1)[s/\cdot] & = & e_1\ e_2 \end{array}$$

Notice that we always prepend the state manipulation to the first computation in the term in the case of let computations.

Lemma 1 (*Substitution*)
The following rules are admissible:

$$\frac{S_1 \vdash^{\mathsf{s}} s_1 : S_2 \qquad S_2 \vdash^{\mathsf{s}} s_2 : S_3}{S_1 \vdash^{\mathsf{s}} s_1.s_2 : S_3}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} e_1 : A \qquad \Gamma, x : A, \Gamma' \vdash^{\mathsf{v}} e_2 : B}{\Gamma, \Gamma' \vdash^{\mathsf{v}} e_2[e_1/x] : B}$$

and

$$\frac{\Gamma \vdash^{\mathsf{v}} e : A \qquad S_1 \vdash^{\mathsf{s}} s : S_2 \qquad \Gamma, x : A, \Gamma'; S_2 \vdash^{\mathsf{c}} c : B; S_3}{\Gamma, \Gamma'; S_1 \vdash^{\mathsf{c}} c[e/x][s/\cdot] : B; S_3}$$

Proof
The state calculus rule is an easy induction. For the value and command calculi, we first prove that the rules

$$\frac{\Gamma \vdash^{\mathsf{v}} e_1 : A \qquad \Gamma, x : A, \Gamma' \vdash^{\mathsf{v}} e_2 : B}{\Gamma, \Gamma' \vdash e_2[e_1/x] : B} \qquad \frac{\Gamma \vdash^{\mathsf{v}} e : A \qquad \Gamma, x : A, \Gamma'; S_1 \vdash^{\mathsf{c}} c : B; S_2}{\Gamma, \Gamma'; S_1 \vdash^{\mathsf{c}} c[e/x] : B; S_2}$$

are admissible by mutual induction on the derivations of the second premises. This gives the value substitution rule in the lemma statement. We then prove the full computation substitution rule admissible by induction on the derivation of the third premise. □

3.2 Examples

3.2.1 Composable Continuations

We present a short example of composable continuations expressed in our calculus, adapted from Wadler's paper (Wadler, 1994). In this case the set of primitive state types is equal to the set of all value types. The operators *reset* and *shift* can be represented as new constructs in the calculus, with the typing rules:

$$\frac{\Gamma; A \vdash^{\mathsf{c}} c : B; B}{\Gamma; C \vdash^{\mathsf{c}} \text{reset } c : A; C} \qquad \frac{\Gamma, f : (T; D) \to (A; D); B \vdash^{\mathsf{c}} c : O; O}{\Gamma; B \vdash^{\mathsf{c}} \text{shift } f.c : T; A.}$$

It is also possible to represent these as primitive commands operating on values of function type, but this method gives a clearer presentation of the example.

As explained above, the intuition behind the shift and reset operators is that "reset" dynamically delimits a context within the execution of the program. The "shift" operation makes this context available to the program as a function, and afterwards returns control to the context outside the enclosing "reset".

An example term is (assuming primitive value functions for numerals and addition):

$$
\begin{array}{l}
\mathrm{let}\ x \Leftarrow \mathrm{reset} \\
\quad\quad (\mathrm{let}\ y \Leftarrow \mathrm{shift}\ f.(\mathrm{let}\ a \Leftarrow f\ 100\ \mathrm{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathrm{let}\ b \Leftarrow f\ 1000\ \mathrm{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (a+b;\cdot)) \\
\quad\quad\ \mathrm{in}\ (y+10;\cdot)) \\
\mathrm{in}\ (1+x;\cdot)
\end{array}
$$

Given the interpretation of composable continuations above, this term evaluates to 1121. The context let $y \Leftarrow -$ in $(y+10;\cdot)$ is invoked twice by the application of the delimited continuation exposed as f by the shift operator. We have used $\cdot$ here to represent the identity in each place where a term of the state calculus may be used.

3.2.2 Session Types

For the session types example, we have two families of operations for each input/output capable type: operations that output a value, given a following context; and operations that input a value, given a following context. We can type these like so:

$$
\frac{\Gamma \vdash^{\mathsf{v}} e : X}{\Gamma; !X.S \vdash^{\mathsf{c}} \mathrm{output}_{X,S}\ e : 1; S}
\qquad\qquad
\frac{}{\Gamma; ?X.S \vdash^{\mathsf{c}} \mathrm{input}_{X,S} : X; S}
$$

Again, we have opted to extend the calculus rather than give these as primitive commands to give a clearer presentation of the example. Assuming we also extend the calculus with a simple if-then-else construct, an example term with its typing is the following:

$$
\begin{array}{l}
-; ?\mathit{Int}.?\mathit{Int}.(!\mathit{Int}.\circ + \circ) \vdash^{\mathsf{c}} \\
\quad \mathrm{let}\ x \Leftarrow \mathrm{input}_{\mathit{Int},?\mathit{Int}.(!\mathit{Int}.\circ+\circ)}\ \mathrm{in} \\
\quad \mathrm{let}\ y \Leftarrow \mathrm{input}_{\mathit{Int},!\mathit{Int}.\circ+\circ}\ \mathrm{in} \\
\quad \mathrm{let}\ z \Leftarrow (x+y;\cdot)\ \mathrm{in} \\
\quad\quad \mathrm{if}\ z > 10 \\
\quad\quad \mathrm{then}\ \mathrm{let}\ \cdot \Leftarrow (\star_1; !\mathit{Int}.\circ + \circ \Rightarrow !\mathit{Int}.\circ)\ \mathrm{in}\ \mathrm{output}_{\mathit{Int},\circ}\ z \\
\quad\quad \mathrm{else}\ (\star_1; !\mathit{Int}.\circ + \circ \Rightarrow \circ) \\
\quad : 1; \circ
\end{array}
$$

where $!\mathit{Int}.\circ + \circ \Rightarrow !\mathit{Int}$ and $!\mathit{Int}.\circ + \circ \Rightarrow \circ$ are primitive state manipulations witnessing the ordering on sessions as defined above.

This term inputs two integers and outputs their sum if it is greater than 10, otherwise it does no output. Its behaviour with respect to input and output is recorded in the state type assigned. Note that we have had to explicitly give the following actions on every input/output operation. This is obviously impractical for any kind of realistic programming. We rectify this problem in Section 4.

3.2.3 Typed State

The following example further demonstrates the difficulties with the need to explicitly declare the context for every effectful operation:

$$
\begin{array}{l}
-; X \times X \vdash^{\mathsf{c}} \\
\quad \text{let } f_1 \Leftarrow \lambda v : \mathit{Int}; X \times X.\text{store}_{\underline{X} \times X}\ v \text{ in} \\
\quad \text{let } f_2 \Leftarrow \lambda v : \mathit{Int}; \mathit{Int} \times X.\mathit{store}_{\mathit{Int} \times \underline{X}}\ v \text{ in} \\
\quad \text{let } \_ \Leftarrow f_1\ 10 \text{ in} \\
\quad \text{let } \_ \Leftarrow f_2\ 20 \text{ in} \\
\quad\quad \star_1 : 1; \mathit{Int} \times \mathit{Int}
\end{array}
$$

This program fragment defines two functions, named f_1 and f_2, that take integer arguments and update the first and second store cells respectively (we use the underline notation to indicate which cell is being mutated). The rest of the program invokes these functions to store the values 10 and 20.

Note that the implementation of these functions has to explicitly name the types of the *entire* state while updating, this is despite the fact that both functions do the same operation: mutate a cell containing an X to a cell containing an integer. Even worse, the order in which these functions are called is fixed: f_1 requires a state of type $X \times X$ while f_2 requires a state of type $\mathit{Int} \times X$, so we must run f_1 first. Finally, if we wish to embed this program inside a larger one that operates on more memory cells, we must rewrite the program to explicitly mention these extra cells.

We will fix all of these problems in Section 4 by allowing commands to be lifted by arbitrary additional state contexts.

3.3 Equational Theory

Equations are generated by three sets of typed axioms of the form $\Gamma \vdash^{\mathsf{v}} e_1 \overset{ax}{=} e_2 : A$, $\Delta \vdash^{\mathsf{s}} s_1 \overset{ax}{=} s_2 : S$ and $\Gamma; \Delta \vdash^{\mathsf{c}} c_1 \overset{ax}{=} c_2 : A; S$, where both sides of each axiom are typable with the given context and type, and the rules in Figure 2, plus reflexivity, transitivity, symmetry and congruence. The state calculus has no additional rules.

The value calculus has the standard $\beta\eta$ rules for product and unit types, plus an η expansion rule for functions. The command calculus incorporates value and state equations via the $(e; s)$ term construct. It also has $\beta\eta$ rules for sequencing, a β rule for functions and commuting conversion for the sequencing construct.

The rules generate three types of equational judgments of the form $\Gamma \vdash^{\mathsf{v}} e_1 = e_2 : A$, $S_1 \vdash^{\mathsf{s}} s_1 = s_2 : S_2$ and $\Gamma; S_1 \vdash^{\mathsf{c}} c_1 = c_2 : A; S_2$. Note that the equations only apply when both sides are well-typed with the same context and result type.

3.4 Interpretation in Parameterised Freyd Categories

The interpretation of the Typed Command Calculus in a parameterised Freyd category has already been alluded to, but we now spell it out in a more detail here.

Value calculus:

$$
\begin{array}{rcl}
\pi_i(e_1, e_2) & = & e_i \\
e & = & (\pi_1 e, \pi_2 e) \\
e & = & \star \\
f & = & \lambda(x^A; S_1).fx
\end{array}
$$

Command calculus rules:

$$
\frac{e_1 = e_2 \qquad s_1 = s_2}{(e_1; s_1) = (e_2; s_2)}
$$

$$
\begin{array}{rcl}
\text{let } x \Leftarrow (e; s) \text{ in } c & = & c[e/x][s/\cdot] \\
\text{let } x \Leftarrow c \text{ in } (x; \cdot) & = & c \\
(\lambda(x^A; S).c)\ a & = & \text{let } x \Leftarrow a \text{ in } c \\
\text{let } x \Leftarrow (\text{let } y \Leftarrow c_1 \text{ in } c_2) \text{ in } c_3 & = & \text{let } y \Leftarrow c_1 \text{ in let } x \Leftarrow c_2 \text{ in } c_3
\end{array}
$$

Fig. 2. Equational Rules for the Typed Command Calculus

Assume a closed parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$, with maps specifying the interpretation of primitive value and state types as $\mathcal{C}$ and $\mathcal{S}$ objects respectively, and the interpretation of primitive value, state manipulation and command operations as arrows in $\mathcal{C}$, $\mathcal{S}$ and $\mathcal{K}$ respectively. The interpretation of types is straightforward.

The state calculus is interpreted in $\mathcal{S}$, using the identity for the S-Id rule and the interpretation of primitive functions, plus composition, for the interpretation of S-Prim. The rules V-Var, V-Prim, V-1I, V-×I and V-×E-i are given the standard interpretation in a category with finite products. The function abstraction rule is interpreted using the isomorphism of homsets derived from the adjunction forming the closure: $\Lambda : \mathcal{K}((\Gamma \times A, S_1), (B, S_2)) \to \mathcal{C}(\Gamma, (A, S_1) \to (B, S_2))$.

The C-V-S rule is interpreted using the functor J in the evident way, and C-Prim is interpreted just using composition. For sequencing, C-Let is interpreted using the premonoidal structure of the parameterised Freyd category. Assuming the first premise is interpreted by an arrow c_1 and the second by an arrow c_2, the conclusion is interpreted by $J(\langle id, id\rangle, id); \Gamma \otimes_{\mathcal{C}} c_1; c_2$. Thus, the context is duplicated using the finite product structure of $\mathcal{C}$, the computation c_1 is executed in context Γ and the result and the remaining copy of Γ are fed into c_2. The conditions on the state types in the rule ensure that the composition is valid. The C-→E rule is interpreted by using the counit of the adjunction forming the closed structure: $ev : ((A, S_1) \to (B, S_2) \times A, S_1) \to (B, S_2)$.

Theorem 2 (*Soundness and Completeness*)
The Typed Command Calculus is sound and complete for closed parameterised Freyd categories.

Proof
Soundness is by induction on the derivations of equational judgments. Completeness is proved by the construction of a closed parameterised Freyd category from the three calculi and the construction of a model within it. See (Atkey, 2006) for the

more general case of the monoidal Typed Command Calculus (Section 5 below). □

4 Structured Parameterisation

In some of the examples presented above we have run into situations where we have been forced to explicitly give a description for the whole state, even when applying an operation that only acts upon a small part. In the typed side-effects example (Sections 2.3.2 and 3.2.3), each of the *read* and *store* operations only acts upon individual memory cells, but the operation itself must mention all the memory cells it does not touch. Likewise, in the session types example (Sections 2.3.4 and 3.2.2), each of the operations must explicitly state all the events that are expected to follow. This makes writing programs intolerably difficult as we must always keep in mind the whole context that a program operates in when working on a small part. This problem is related to the *frame problem* of Artificial Intelligence; we have to explicitly declare everything that is *not* affected by an operation, as well as the things that are.

The solution we propose here is to make use of the structure present in the state descriptions used in our examples and lift this structure up to the level of computations. In the typed side-effects example, states consisting of multiple memory cells are constructed using symmetric monoidal structure. Thus we have operations $S \otimes -$ and $- \otimes S$ for building composite state descriptions. We lift these structuring operations to the level of computations by requiring natural transformations $(- \otimes S)^\dagger : T(S_1, S_2, A) \to T(S_1 \otimes S, S_2 \otimes S, A)$, and symmetrically. The idea is that $(- \otimes S)^\dagger$ takes a computation that operates "locally" and lifts it up to a larger context. The meaning of "local" here is similar to the local reasoning of Separation Logic (O'Hearn *et al.*, 2001). Indeed, the natural transformation $(- \otimes S)^\dagger$ is reminiscent of the frame rule of Separation Logic:

$$\frac{\{P\}c\{Q\}}{\{P * S\}c\{Q * S\}}$$

In this rule, the specification $\{P\} - \{Q\}$ has been proved "locally" about some program c. This is then lifted to a larger context by adjoining an additional state description S. We follow up this example in Section 4.2.2 by considering a minimal version of Separation Logic in terms of parameterised monads.

We also note that the strength τ of a (parameterised) monad also serves to interpret the lifting of computations up to a larger context. The similarity between the two actions will become more apparent when we consider the extension of parameterised Freyd categories to structured parameterisation in Section 4.3.

4.1 Endofunctor Liftings for Parameterised Monads

We assume that the relevant structure we require has been given as endofunctors and natural transformations on the category $\mathcal{S}$ of state descriptions and manipulations. For example, symmetric monoidal structure is given as a family of pairs of endofunctors $S \otimes -$ and $- \otimes S$ that together are bifunctorial with the associated associativity, left and right unit and symmetry natural transformations. The following definition describes the requirements on a suitable lifting of this structure to the level of computations modelled by an $\mathcal{S}$-parameterised monad.

Definition 5

Given an $\mathcal{S}$-parameterised monad (T, η, μ), and a functor $F : \mathcal{S} \to \mathcal{S}$, a *lifting* of F to T is a natural transformation $F^{\dagger}_{S_1,S_2,A} : T(S_1, S_2, A) \to T(FS_1, FS_2, A)$ that commutes with the unit, multiplication and strength of the monad:

$$
\begin{tikzcd}
T(S_1, S_2, T(S_2, S_3, A)) \arrow[r, "F^{\dagger}"] \arrow[d, "\mu"'] & T(FS_1, FS_2, T(S_2, S_3, A)) \arrow[d, "T(FS_1,FS_2,F^{\dagger})"] \\
T(S_1, S_3, A) \arrow[dr, "F^{\dagger}"'] & T(FS_1, FS_2, T(FS_2, FS_3, A)) \arrow[d, "\mu"] \\
& T(FS_1, FS_3, A)
\end{tikzcd}
$$

$$
\begin{tikzcd}
A \arrow[r, "\eta"] \arrow[dr, "\eta_F"'] & T(S, S, A) \arrow[d, "F^{\dagger}"] \\
& T(FS, FS, A)
\end{tikzcd}
\qquad
\begin{tikzcd}
A \times T(S_1, S_2, B) \arrow[r, "\tau"] \arrow[d, "A \times F^{\dagger}"'] & T(S_1, S_2, A \times B) \arrow[d, "F^{\dagger}"] \\
A \times T(FS_1, FS_2, B) \arrow[r, "\tau"] & T(FS_1, FS_2, A \times B)
\end{tikzcd}
$$

A natural transformation $\zeta : F \Rightarrow G$ in $\mathcal{S}$ is *natural for liftings* $F^{\dagger}$ and $G^{\dagger}$ if the diagram

$$
\begin{tikzcd}
T(S_1, S_2, A) \arrow[r, "F^{\dagger}"] \arrow[d, "G^{\dagger}"'] & T(FS_1, FS_2, A) \arrow[d, "T(FS_1,\zeta,A)"] \\
T(GS_1, GS_2, A) \arrow[r, "T(\zeta,GS_2,A)"] & T(FS_1, GS_2, A)
\end{tikzcd}
$$

commutes.

Extending the alternative partial definition of a parameterised monad as a $\mathcal{C}^{\mathcal{C}}$-enriched category noted above, the definition of a lifting of a functor is the same as a $\mathcal{C}^{\mathcal{C}}$-functor on this category.

Lemma 2

If we have two natural transformations $\zeta : F \Rightarrow G$ and $\zeta' : G \Rightarrow H$ that are natural for liftings $F^{\dagger}$, $G^{\dagger}$ and $H^{\dagger}$ then their composition $\zeta; \zeta'$ is natural for $F^{\dagger}$ and $H^{\dagger}$.

Proof

Consider the following diagram, where the outer diagram is the one we want to commute.

$$
\begin{tikzcd}
T(S_1, S_2, A) \arrow[rrr, "F^\dagger"] \arrow[dr, "G^\dagger"] \arrow[ddd, "H^\dagger"'] & & & T(FS_1, FS_2, A) \arrow[dl, "T(FS_1,\zeta,A)"'] \arrow[ddd, "T(FS_1,\zeta;\zeta',A)"] \\
& T(GS_1, GS_2, A) \arrow[r, "T(\zeta,GS_2,A)"] \arrow[d, "T(GS_1,\zeta',A)"'] & T(FS_1, GS_2, A) \arrow[ddr, "T(FS_1,\zeta',A)"] & \\
& T(GS_1, HS_2, A) \arrow[drr, "T(\zeta,HS_2,A)"'] & & \\
T(HS_1, HS_2, A) \arrow[ur, "T(\zeta',HS_2,A)"] \arrow[rrr, "T(\zeta;\zeta',HS_2,A)"'] & & & T(FS_1, HS_2, A)
\end{tikzcd}
$$

The internal diagrams all commute: the top-most and left-most commute since ζ and ζ' are natural for the liftings of the functors; the bottom-most and right-most commute since T is a functor so it preserves composition; and the centre diagram commutes by the bifunctoriality of T. Hence the outer diagram commutes. $\square$

Using Definition 5, we can state the structure we require on parameterised monads to interpret structured parameterisation for our examples.

Definition 6
An $\mathcal{S}$-parameterised monad (T, η, μ) has monoidal lifting if, for every $S \in \mathsf{Ob}\mathcal{S}$, there are liftings for the functors $- \otimes S$ and $S \otimes -$, written $(- \otimes S)^\dagger$ and $(S \otimes -)^\dagger$, such that all the monoidal structure transformations – associativity and left and right unit – are natural for them and so are the natural transformations given by $- \otimes s$ and $s \otimes -$ for every arrow s. The monad has symmetric monoidal lifting if the symmetry natural transformations are also natural.

4.2 Examples

We now take some of the examples from Section 2.3 and show how the addition of structured parameterisation helps.

4.2.1 Typed Side-effects

On the $\mathcal{C}$-parameterised monad defined in Section 2.3.2 above, we can define symmetric monoidal liftings as:

$$(A \times -)^\dagger = c \mapsto \lambda(s, s_1).\text{let } (s_2, a) = c(s_1) \text{ in } ((s, s_2), a)$$

and similarly for $(- \times A)^\dagger$. These lift the finite product structure of $\mathcal{C}$ up to computations. With these definitions we do not need to explicitly give the whole state for the *read* and *store* operations, they can be lifted up to the required contexts by these operations.

4.2.2 Minimal Separation Logic

The *read* and *store* operations of previous example operate on specific store cells, where the cell selected for each operation is determined by the use of the symmetric monoidal lifting operations. An alternative is to annotate each read and store with the abstract location of the heap cell upon which it operates.

For this example we will use a cut-down variant of Separation Logic (O'Hearn *et al.*, 2001) for the state descriptions – only the type of the contents of memory cells is recorded. Entailment of assertions as the arrows of the state category. Computations in the parameterised monad will be "local" commands that satisfy Separation Logic specifications.

Assume some set L of locations. Stores are then partial maps from locations to values, which we take in this example to be either integers or booleans: $St = L \rightharpoonup \mathbb{Z} + \mathbb{B}$. Two stores are separate ($s_1 \# s_2$) if $\mathrm{dom}(s_1) \cap \mathrm{dom}(s_2) = \emptyset$. We define a partial operation of store combination by:

$$s_1 * s_2 = \begin{cases} l \mapsto \begin{cases} s_1(l) & \text{if } l \in \mathrm{dom}(s_1) \\ s_2(l) & \text{if } l \in \mathrm{dom}(s_2) \end{cases} & \text{if } s_1 \# s_2 \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The language of assertions is given by the grammar:

$$S ::= l \mapsto \mathbb{Z} \mid l \mapsto \mathbb{B} \mid l \mapsto ? \mid S_1 * S_2$$

The first three assertion kinds state that a store consists of a single cell l containing an integer, a boolean or a indeterminate value respectively. The final kind asserts that the store consists of two separate sub-stores described by S_1 and S_2 respectively. This semantics is formalised by the following definition of satisfaction:

$$\begin{array}{lllll} s & \models & l \mapsto \mathbb{Z} & \text{iff} & \exists i \in \mathbb{Z}.\ s = \{l \mapsto i\} \\ s & \models & l \mapsto \mathbb{B} & \text{iff} & \exists b \in \mathbb{B}.\ s = \{l \mapsto b\} \\ s & \models & l \mapsto ? & \text{iff} & \exists v \in \mathbb{Z} + \mathbb{B}.\ s = \{l \mapsto v\} \\ s & \models & S_1 * S_2 & \text{iff} & \exists s_1, s_2.\ s_1 * s_2 \simeq s \wedge s_1 \models S_1 \wedge s_2 \models S_2 \end{array}$$

The relation $\simeq$ is true iff both sides are defined and equal, or both sides are undefined. We define entailment between assertions as $S_1 \models S_2$ iff for all s. $s \models S_1$ implies $s \models S_2$. We treat assertions and entailment as a symmetric monoidal category, the symmetric monoidal functor given by $S_1 * S_2$.

Following (O'Hearn *et al.*, 2004), we define local commands as a subset of side-effecting commands that can fault. For a set A, define $LCom(A)$ as elements $c \in St \to ((St \times A) + \{\mathsf{fault}\})$ that satisfy a locality condition:

Locality For all s, s_1, s' and a such that $s \# s_1$, if $c(s) = (s', a)$ then $s' \# s_1$ and $c(s * s_1) = (s' * s_1, a)$.

This condition states that if a command completes successfully (i.e. does not result in fault) for a store s, then if we attach any additional store s_1, then this store is preserved and the result is the same as before. The key idea is that a command will fault if it is provided a store that does not contain the locations it requires. We can get away with a simpler functional description of commands here instead of the relational description in (O'Hearn *et al.*, 2004) because we do not consider non-deterministic memory allocation – our aim is to show how parameterised monads can handle locality.

We can now give the description of our parameterised monad. The functor part is defined as:

$$T(S_1, S_2, A) = \{c \in LCom(A) \mid \forall s.\ s \models S_1 \Rightarrow \exists s', a.\ c(s) = (s', a) \wedge s' \models S_2\}$$

So computations are local commands that obey a specification for their start and end states. The unit and multiplication are defined as for the traditional state monad. We must check that these operations introduce and preserve locality, but this is straightforward.

There are two primitive operations for this monad: reading values of type A from a location and storing new values of type A at a given location, where $A \in \{\mathbb{Z}, \mathbb{B}\}$:

$$\begin{array}{rcl} read_{l,A} & : & T(l \mapsto A, l \mapsto A, A) \\ read_{l,A} & = & \lambda s. \begin{cases} (s, s(l)) & \text{if } l \in \mathrm{dom}(s) \\ \mathsf{fault} & \text{otherwise} \end{cases} \\ \\ store_{l,A} & : & A \to T(l \mapsto ?, l \mapsto A, 1) \\ store_{l,A} & = & a \mapsto \lambda s. \begin{cases} (s[l \mapsto a], \star) & \text{if } l \in \mathrm{dom}(s) \\ \mathsf{fault} & \text{otherwise} \end{cases} \end{array}$$

Thus, reading from l looks up that location in the store and returns the value stored there, faulting if it is not present. Storing updates the store at l, faulting if the location is not in the current store. Both of these operations are clearly local and match their given specifications.

Finally, we define the liftings of the assertion's symmetric monoidal structure. Due to the locality property we have required on computations, this is just inclusion: $T(S_1, S_2, A) \subseteq T(S_1 \otimes S, S_2 \otimes S, A)$, and symmetrically. Locality ensures that computations act the same in larger stores.

Our definition of *read* and *store* hard-code the locations that a program accesses into its program text. We discuss in Section 6 the possibility of using indexing to relax this restriction and still retain the varying of types of reference cells over the execution of the program.

4.2.3 Categories

Recall that the parameterised monad in this example is $T_{\mathcal{S}_1}(S_1, S_2, A) = \mathcal{S}(S_1, S_2) \times A$. Given any functor $F : \mathcal{S}_1 \to \mathcal{S}_1$ that is also a functor on the parameterising subcategory $\mathcal{S}$, there is an obvious lifting $F^\dagger$ defined as $F^\dagger(s, a) = (Fs, a)$. Natural transformations $\zeta : F \Rightarrow G$ are automatically natural for these liftings.

In the case of the category StkPrg we have a natural monoidal structure given by addition on the objects. With the liftings of this monoidal structure, we need no longer provide the depth of the current stack for each of the basic operations of the monad:

$$\begin{array}{rcl} push : \mathbb{Z} \to T_{\mathrm{StkPrg}}(0,1,1) & = & i \mapsto ([\mathsf{push}.i], \star) \\ add : 1 \to T_{\mathrm{StkPrg}}(2,1,1) & = & \star \mapsto ([\mathsf{add}], \star) \\ dup : 1 \to T_{\mathrm{StkPrg}}(1,2,1) & = & \star \mapsto ([\mathsf{dup}], \star) \end{array}$$

4.2.4 Typed I/O: Session Types

In this example, the state description category has, for any session type S, an endofunctor $-.S$ given by substitution for $\circ$. We can define a lifting for the functor $-.S$ by induction over the tree structure of $T_\Omega(S_1, S_2, A)$: for each Ω-operation $\mathrm{input}_{X,S'}$ or $\mathrm{output}_{X,S'}$ in the tree, there exists an Ω-operation $\mathrm{input}_{X,S'.S}$ and $\mathrm{output}_{X,S'.S}$. Hence we get a lifting

$$(-.S)^\dagger : T_\Omega(S_1, S_2, A) \to T_\Omega(S_1.S, S_2.S, A).$$

Hence, we can give the primitive monad operations as

$$\begin{array}{rcl} \mathrm{input}_X & : & 1 \to T(?X, \circ, X) \\ \mathrm{output}_X & : & X \to T(!X, \circ, 1) \end{array}$$

and rely on the lifting to append future traces as needed. Now, if we have computations represented by arrows $c_1 : A \to T(?X, \circ, B)$ and $c_2 : B \to T(!X, \circ, C)$ then we can sequence then using the lifting:

$$\begin{tikzcd} A \arrow[r, "c_1"] & T(?X, \circ, B) \arrow[r, "(-.!X)^\dagger"] & T(?X.!X, !X, B) \arrow[d, "T(?X.!X,!X,c_2)"] & \\ & & T(?X.!X, !X, T(!X, \circ, C)) \arrow[r, "\mu"] & T(?X.!X, \circ, C) \end{tikzcd}$$

4.2.5 Monads with a Single Parameter

Given the operations in the previous example, a natural question is whether our style of parameterisation is required in the case of session types. It would seem that a monad with a single parameterisation giving the session carried out by the computation would suffice. That is, instead of $T(!X.?X.\circ, \circ, A)$, one would just have $T(!X.?X, A)$. We now briefly discuss this alternative definition. Wadler and Thiemann (Wadler & Thiemann, 2003) investigated the link between monads and effect types by focusing on the indexing of monads by a single parameter.

We assume some base category $\mathcal{C}$ and a strict monoidal category of "effect types" $\mathcal{E}$ with unit $\emptyset$ and monoidal bifunctor $\varepsilon \cdot \varepsilon'$. The functor part of the monad has type $T : \mathcal{E}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$, the idea being that $T(\varepsilon, A)$ describes computations that do effects described by ε, yielding values of type A. The functorial action on the first argument provides for sub-effecting. The unit and multiplication are natural transformations with types:

$$\begin{array}{rcl} \eta_A & : & A \to T(\emptyset, A) \\ \mu_{\varepsilon,\varepsilon',A} & : & T(\varepsilon, T(\varepsilon', A)) \to T(\varepsilon \cdot \varepsilon', A) \end{array}$$

The unit provides a computation that performs no effects , and the multiplication sequences two computations, combining their effect annotations.

Given a singly-parameterised monad $T : \mathcal{E}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$, it is easy to express it by means of a (doubly-)parameterised monad $T' : \mathcal{E}^{\mathsf{op}} \times \mathcal{E} \times \mathcal{C} \to \mathcal{C}$ by setting $T(\varepsilon, A) = T'(\varepsilon, \emptyset, A)$, with unit given by $\eta'_\emptyset$ and multiplication as for the session types example above. We read this as interpreting computations with effects ε as computations that start with the potential to do the effects in ε and end with no potential. Thus, given a type system with a single effect parameter, to give a semantics, it suffices to look for a parameterised monad as we have defined it.

Going in the opposite direction, it is not clear how to proceed. There does not seem to be an obvious way to combine the two parameters of a parameterised monad into the single parameter of the definition in this section in such a way that interacts well with the multiplication and unit. We take this mismatch to mean that parameterised monads provide a more refined view of effectful computations since they can speak directly about the state before and after the computation.

As a further argument in favour of our definition is that the obvious definition of Kleisli category for a singly-parameterised monad requires a new kind of category, where the homsets are fibred over $\mathcal{E}$. That is, for each pair of objects of $\mathcal{C}$, A and B there is a function $\mathit{eff} : \mathcal{C}_T(A, B) \to \mathsf{Ob}\mathcal{E}$, together with reindexing functions $f^* : \varepsilon_2^* \to \varepsilon_1^*$ for every arrow $f : \varepsilon_1 \to \varepsilon_2$ in $\mathcal{E}$, where $\varepsilon^* = \{g \in \mathcal{C}_T(A, B) \mid \mathit{eff}(g) = \varepsilon\}$. The identities must satisfy $\mathit{eff}(id) = \emptyset$ and composition must satisfy $\mathit{eff}(f; g) = \mathit{eff}(f) \cdot \mathit{eff}(g)$. Such a definition already brings complications by stepping outside usual category theory, and it is unclear, to this author at least, what a suitable definition of adjunction between such a category and a normal category is.

4.2.6 Typed I/O: Multiple independent I/O channels

The typed I/O construction can be extended to monoidal parameterisation. This can be used to model the use of multiple independent I/O devices. Given a discrete category $\mathcal{S}$ of state descriptions, and a collection of operations Ω, as defined above, we define a new monad parameterised by the free strict monoidal category on $\mathcal{S}$. The idea is that an object represents an array of devices in their respective states. The notation $S(S')$ denotes an object of this category with a distinguished location holding an $\mathcal{S}$ object S'. We construct a monad T_{Ω^*}. On objects, it is built from the following rules:

$$\frac{a \in A}{\mathrm{e}(a) \in T_{\Omega^*}(S, S, A)}$$

$$\frac{op \in \Omega \qquad o \in \mathsf{out}(op) \qquad k \in \mathsf{in}(op) \to T_{\Omega^*}(S(\mathsf{post}(op)), S', A)}{S(op)(o, k) \in T_{\Omega^*}(S(\mathsf{pre}(op)), S', A)}$$

This construction is subject to the smallest equivalence relation that respects the $S(op)(o, -)$ operations and including the following equation, given that $S(-) \neq$

$S'(-)$:

$$S(op)(o, \lambda i.\ S'(op')(o', \lambda i'.\ k\ i\ i')) = S'(op')(o', \lambda i'.\ S(op)(o, k\ i\ i'))$$

Therefore, computations are trees of input/output operations-in-context that branch for inputs, with values at the leaves. The parameterising category is discrete, so we do not have to define the monad on any state manipulation arrows. Note that $T(S, S', A)$ is empty if S and S' are of different sizes – we cannot throw I/O devices away, or generate new ones. The equation states that operations on independent devices in different slots are independent and can be commuted past each other. Monad unit and multiplication are defined as above. Monoidal lifting is defined by appending additional context to the left or right of each node.

4.3 Structured Parameterisation for Freyd Categories

Definition 7
A parameterised Freyd category $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ has a lifting of an endofunctor $F : \mathcal{S} \to \mathcal{S}$ if it has a functor $F^\star : \mathcal{K} \to \mathcal{K}$ such that $F^\star(J(A, S)) = J(A, FS)$ and $F^\star(J(f, s)) = J(f, Fs)$. A natural transformation $\zeta : F \Rightarrow G$ is natural for liftings $F^\star$ and $G^\star$ if the diagram

$$\begin{tikzcd}
J(A, FS) \arrow[r, "J(A,\zeta)"] \arrow[d, "F^* f"'] & J(A, GS) \arrow[d, "G^* f"] \\
J(B, FS') \arrow[r, "J(B,\zeta)"] & J(B, GS')
\end{tikzcd}$$

commutes for all $f : (A, S) \to (B, S')$. This lifting must commute with the parameterised Freyd structure:

$$\begin{aligned}
F^*(A \ltimes_{\mathcal{C}} (B, S)) &= A \ltimes_{\mathcal{C}} (B, FS) \\
F^*(f \ltimes_{\mathcal{C}} c) &= f \ltimes_{\mathcal{C}} (F^* c)
\end{aligned}$$

and similarly for $\rtimes_{\mathcal{C}}$.

In the case of monoidal liftings, when the endofunctors are $S \otimes -$ and $- \otimes S$ and the natural transformations are the associativity, left and right unit and symmetry, the conditions required here are exactly the same as for premonoidal structure with respect to $\mathcal{C}$, except that it is with respect to $\mathcal{S}$. In this special case, the definition is somewhat more symmetric than that for parameterised monads. This is to be expected, given that the focus of the definition of (parameterised) Freyd category is directly upon computation in context, so it easier to extend the definition to multiple premonoidal structures, and so multiple kinds of computation in context. We call the special case of symmetric monoidal lifting a *double* parameterised Freyd category.

Theorem 3
For an $\mathcal{S}$-parameterised monad (T, η, μ) on $\mathcal{C}$, given a lifting $F^\dagger$ of an endofunctor

$F : \mathcal{S} \to \mathcal{S}$, we get a lifting $F^\star$ on the parameterised Freyd category $\mathcal{C}_T$ and *vice versa*. These operations are inverse. Moreover, given a natural transformation $\zeta : F \Rightarrow G$ that is natural for liftings $F^\dagger$ and $G^\dagger$, then it is also natural for liftings $F^\star$ and $G^\star$, and *vice versa*.

Proof
Given a lifting $F^\dagger$ on the parameterised monad T, define the lifting $F^\star$ on $\mathcal{C}_T$ for $F^\star c$ as the composite

$$A \xrightarrow{\quad c \quad} T(S_1, S_2, B) \xrightarrow{F^\dagger} T(FS_1, FS_2, B)$$

Given a lifting $F^\star$ on a parameterised Freyd category, define the lifting $F^\dagger$ on the derived monad using the closed structure of the Freyd category, recalling that the derived monad's functor is $T(S_1, S_2, A) = (1, S_1) \to (A, S_2)$:

$$\cfrac{\cfrac{\mathrm{ev} : ((1, S_1) \to (A, S) \times 1, S_1) \longrightarrow (A, S_2)}{F^\star(\mathrm{ev}) : ((1, S_1) \to (A, S) \times 1, FS_1) \longrightarrow (A, FS_2)}}{\Lambda(F^\star(\mathrm{ev})) : [(1, S_1) \to (A, S_2)] \longrightarrow [(1, FS_1) \to (A, FS_2)]}$$

It is routine to check that both these definitions obey the required axioms. In particular, note that the requirement that $F^\dagger$ commutes with the monad multiplication μ directly corresponds to the requirement that $F^\star$ preserves composition, likewise for commutativity with η and preservation of identities. Also, the requirement that $F^\dagger$ commutes with the strength directly corresponds to preservation of $\otimes_{\mathcal{C}}$ and $\otimes_{\mathcal{C}}$. That they are mutually inverse definitions can be seen by writing out the definitions and calculating, keeping in mind the differences in composition in $\mathcal{C}$ and $\mathcal{C}_T$. Checking that these operations preserve the naturality of natural transformations on $\mathcal{S}$ is also routine. □

5 Symmetric Monoidal Typed Command Calculus

Given the wide range of possible structures possible on the state category $\mathcal{S}$, it is infeasible to give a neat calculus that covers all of them. Therefore, we focus on a single important example: that of symmetric monoidal structure. In this section we extend the calculus of Section 3 so that it is sound and complete for closed double parameterised Freyd categories. We call the extended calculus the Symmetric Monoidal Typed Command Calculus. The changes to the typing rules are shown in Figure 3. The terms, types and rules for the value calculus are unchanged, except by the larger range of state type constructors:

$$S ::= X \in \mathcal{T}_S \mid I \mid S_1 \otimes S_2$$

State contexts are now lists of state manipulation variables and state type pairs, ranged over by Δ and with the condition that no variable appears more than once. We define the merging relation $- \bowtie - \approx -$ on triples of contexts by the rules:

$$\cfrac{}{I \bowtie I \approx I} \qquad \cfrac{\Delta_1 \bowtie \Delta_2 \approx \Delta}{(\Delta_1, z : S) \bowtie \Delta_2 \approx \Delta, z : S} \qquad \cfrac{\Delta_1 \bowtie \Delta_2 \approx \Delta}{\Delta_1 \bowtie (\Delta_2, z : S) \approx \Delta, z : S}$$

State Calculus:

$$\frac{}{z : S \vdash^{\mathsf{s}} z : S}\ (\text{S-Var}) \qquad \frac{\Delta \vdash^{\mathsf{s}} s : S_1 \qquad (\mathtt{m} : S_1 \longrightarrow S_2) \in \Phi_S}{\Delta \vdash^{\mathsf{s}} \mathtt{m}\ s : S_2}\ (\text{S-Prim})$$

$$\frac{\Delta_1 \vdash^{\mathsf{s}} s_1 : S_1 \qquad \Delta_2 \vdash^{\mathsf{s}} s_2 : S_2}{\Delta_1 \bowtie \Delta_2 \vdash^{\mathsf{s}} (s_1, s_2) : S_1 \otimes S_2}\ (\text{S-}\otimes\text{I})$$

$$\frac{\Delta_1 \vdash^{\mathsf{s}} s_1 : S_1 \otimes S_2 \qquad \Delta_2, z_1 : S_1, z_2 : S_2 \vdash^{\mathsf{s}} s_2 : S_3}{\Delta_1 \bowtie \Delta_2 \vdash^{\mathsf{s}} \text{let } (z_1, z_2) = s_1 \text{ in } s_2 : S_3}\ (\text{S-}\otimes\text{E}) \qquad \frac{}{I \vdash^{\mathsf{s}} \star_I : I}\ (\text{S-}I\text{I})$$

$$\frac{\Delta_1 \vdash^{\mathsf{s}} s_1 : I \qquad \Delta_2 \vdash^{\mathsf{s}} s_2 : S}{\Delta_1 \bowtie \Delta_2 \vdash^{\mathsf{s}} \text{let } \star_I = s_1 \text{ in } s_2 : S}\ (\text{S-}I\text{E})$$

Value Calculus:

$$\frac{x : A \in \Gamma}{\Gamma \vdash^{\mathsf{v}} x : A}\ (\text{V-Var}) \qquad \frac{\Gamma \vdash^{\mathsf{v}} e : A_1 \qquad (\mathtt{f} : A_1 \longrightarrow A_2) \in \Phi_V}{\Gamma \vdash^{\mathsf{v}} \mathtt{f}\ e : A_2}\ (\text{V-Prim})$$

$$\frac{}{\Gamma \vdash^{\mathsf{v}} \star_1 : 1}\ (\text{V-1I}) \qquad \frac{\Gamma \vdash^{\mathsf{v}} e_1 : A_1 \qquad \Gamma \vdash^{\mathsf{v}} e_2 : A_2}{\Gamma \vdash^{\mathsf{v}} (e_1, e_2) : A_1 \times A_2}\ (\text{V-}\times\text{I})$$

$$\frac{\Gamma \vdash^{\mathsf{v}} e : A_1 \times A_2}{\Gamma \vdash^{\mathsf{v}} \pi_i e : A_i}\ (\text{V-}\times\text{E-}i) \qquad \frac{\Gamma, x : A_1; z : S_1 \vdash^{\mathsf{c}} c : A_2; S_2}{\Gamma \vdash^{\mathsf{v}} \lambda(x^{A_1}; z^{S_1}).c : (A_1; S_1) \to (A_2; S_2)}\ (\text{V-}{\to}\text{I})$$

Command Calculus:

$$\frac{\Gamma \vdash^{\mathsf{v}} e : A \qquad \Delta \vdash^{\mathsf{s}} s : S}{\Gamma; \Delta \vdash^{\mathsf{c}} (e; s) : A; S}\ (\text{C-V-S})$$

$$\frac{\Gamma \vdash^{\mathsf{v}} e : A \qquad (\mathtt{p} : (A; S_1) \longrightarrow (B; S_2)) \in \Phi_C}{\Gamma; z : S_1 \vdash^{\mathsf{c}} \mathtt{p}\ (e; z) : B; S_2}\ (\text{C-Prim})$$

$$\frac{\Gamma; \Delta_1 \vdash^{\mathsf{c}} c_1 : A_1; S_1 \qquad \Gamma, x : A_1; \Delta_2, z : S_1 \vdash^{\mathsf{c}} c_2 : A_2; S_2}{\Gamma; \Delta_1 \bowtie \Delta_2 \vdash^{\mathsf{c}} \text{let } (x; z) \Leftarrow c_1 \text{ in } c_2 : A_2; S_2}\ (\text{C-Let})$$

$$\frac{\Gamma; \Delta_1 \vdash^{\mathsf{c}} c_1 : A_1; S_1 \otimes S_2 \qquad \Gamma, x : A_1; \Delta_2, z_1 : S_1, z_2 : S_2 \vdash^{\mathsf{c}} c_2 : A_2; S_3}{\Gamma; \Delta_1 \bowtie \Delta_2 \vdash^{\mathsf{c}} \text{let } (x; z_1, z_2) \Leftarrow c_1 \text{ in } c_2 : A_2; S_3}\ (\text{C-Let-}\otimes)$$

$$\frac{\Gamma; \Delta_1 \vdash^{\mathsf{c}} c_1 : A_1; I \qquad \Gamma, x : A_1; \Delta_2 \vdash^{\mathsf{c}} c_2 : A_2; S_3}{\Gamma; \Delta_1 \bowtie \Delta_2 \vdash^{\mathsf{c}} \text{let } (x; \star_I) \Leftarrow c_1 \text{ in } c_2 : A_2; S_3}\ (\text{C-Let-}I)$$

$$\frac{\Gamma \vdash^{\mathsf{v}} e_1 : (A; S_1) \to (B; S_2) \qquad \Gamma \vdash^{\mathsf{v}} e_2 : A}{\Gamma; z : S_1 \vdash^{\mathsf{c}} e_1(e_2; z) : B; S_2}\ (\text{C-}{\to}\text{E})$$

Fig. 3. Typing rules for the Monoidal Typed Command Calculus

Thus if $\Delta_1 \bowtie \Delta_2 \approx \Delta$ then Δ_1 and Δ_2 have no variables in common. Given contexts Δ_1, Δ_2, we write $\Delta_1 \bowtie \Delta_2$ to stand for any context Δ such that $\Delta_1 \bowtie \Delta_2 \approx \Delta$.

The state calculus has additional rules for introducing and eliminating pair and unit types, following the standard term constructs for substructural calculi. The command calculus retains the rules C-V-S, C-Prim and C-$\to$E.

There are now three sequencing constructs in the command calculus, typed by the rules C-Let, C-Let-$\otimes$ and C-Let-I. All the rules type the execution of a command c_1, lifted up to the context of $(\Gamma; \Delta_2)$, followed by the execution of a second command c_2. The rule C-Let differs in this calculus from the one in the Typed Command Calculus by allowing computation in a state context, as well as in a value context.

The three sequencing rules differ in the de-structuring of the state output of the first term. The C-Let rule does no de-structuring and passes the state output of c_1 directly into c_2. Rule C-Let-$\otimes$ takes a state pair from c_1 and splits it into two separate variables in c_2's context. Rule C-Let-I takes a unit state and discards it. To see why these constructs are needed, consider the following example. Assume we have a primitive command $\mathtt{p} : (1, I) \to (1, S \otimes S)$. We can use this command in a sequencing construct:

$$\text{let } (x; z) \Leftarrow \mathtt{p}(*_1, *_I) \text{ in } \ldots$$

However, without C-Let-$\otimes$ there would be no way to decompose the variable z bound in the body of this expression in a way that would allow us to use the components in two different commands. Assuming two commands c_1 and c_2 with free variables z_1 and z_2 respectively, the use of C-Let-$\otimes$ allows us to use the output of $\mathtt{p}$ in both:

$$\text{let } (x; z_1, z_2) \Leftarrow \mathtt{p}(*_1; *_I) \text{ in let } (x; z_1') \Leftarrow c_1 \text{ in let } (x; z_2') \Leftarrow c_2 \text{ in} \ldots$$

The C-Let-I rule fulfils a similar role in eliminating variables of type I.

5.1 Example and a Variation

We now present an example program in our calculus and discuss an alternative calculus for the semantic structures we have defined.

5.1.1 Example

We rewrite the example from Section 3.2.3 to take advantage of the lifting operations:

$$\begin{array}{l}
-; z : X \times X \vdash^{\mathsf{c}} \\
\quad \text{let } f \Leftarrow \lambda(v^{Int}; z^X).\text{store } (v; z) \text{ in} \\
\quad \text{let } (_; z_1, z_2) \Leftarrow (\star_1; z) \text{ in} \\
\quad \text{let } (_; z_1) \Leftarrow f\ (10; z_1) \text{ in} \\
\quad \text{let } (_; z_2) \Leftarrow f\ (20; z_2) \text{ in} \\
\quad\quad (\star_1; (z_1, z_2)) : 1; \mathit{Int} \times \mathit{Int}
\end{array}$$

Here, we need only write the function to store an integer once. We explicitly pass around the pieces of the state that we are interested in. The pattern match on the second “let” splits the state into two, the two parts are operated on separately by the two invocations of f and then they are put back together for the result.

This explicit manipulation of the state is sometimes useful and sometimes not; in the next example we show how to alter the calculus to make the state part implicit.

5.1.2 Variation: Implicit State Calculus

In the calculus of Figure 3, the state calculus is fully explicit, and in the previous example this is used in order to distribute the parts of the state around the program. This is essential in order to disambiguate which piece of state each *read* and *store* operation acts upon. As we explained in the example in Section 4.2.2, an alternative is to annotate *read* and *store* operations with the memory locations they are acting upon. We can then use a minimal version of Separation Logic to describe the state. We now discuss the relevant changes to the calculus to support the situation when the state category is a partially ordered set with a ordered monoid structure.

The first act is to remove the state calculus altogether and replace it with a single judgment form $S_1 \Rightarrow S_2$ indicating the entailment relation of the assertions. State contexts are replaced with a single assertion, similar to the Typed Command Calculus in Section 3. We change the C-V-S rule to remove the state calculus component:

$$\frac{\Gamma \vdash^{\mathsf{v}} e : A}{\Gamma; S \vdash^{\mathsf{c}} \mathrm{val}_S\ e : A; S}\ (\text{C-V})$$

This rule incorporates a given pure value into the command calculus, at a fixed state type. We incorporate the partial order on state types by a rule of consequence:

$$\frac{S_1 \Rightarrow S_1' \qquad \Gamma; S_1' \vdash^{\mathsf{c}} c : A; S_2' \qquad S_2' \Rightarrow S_2}{\Gamma; S_1 \vdash^{\mathsf{c}} c : A; S_2}\ (\text{C-}\textsc{Conseq})$$

Note that due to the fact there is no term-level syntax associated with the C-Conseq rule, the semantics of this calculus is defined over typing derivations and not terms. This means that for some uses of the semantics in parameterised Freyd categories coherence issues must be addressed, similar to (Birkedal *et al.*, 2006).

We also remove all the sequencing C-Let rules and replace them with a single rule:

$$\frac{\Gamma; S_1 \vdash^{\mathsf{c}} c_1 : A; S_2 \qquad \Gamma, x : A; S_2 * S \vdash^{\mathsf{c}} c_2 : B; S_3}{\Gamma; S_1 * S \vdash^{\mathsf{c}} \mathrm{let}\ x \Leftarrow c_1\ \mathrm{in}\ c_2 : B; S_3}\ (\text{C-}\textsc{Let})$$

This is semantically identical to the old C-Let rule. The only difference is that there is a single state type on the left side of the judgment, rather than a context. It is similar to the C-Let rule of the Typed Command Calculus in Section 3, except that we allow an additional state type S, that c_1 is unaware of, to be passed to c_2.

We also alter the C-Prim, C-$\to$E and V-$\to$I rules to remove the variable name from the state context.

Using the location-annotated read and write operations from the example in Section 4.2.1, we can write the following program in the new calculus:

$$\begin{array}{l} -; l_1 \mapsto \mathbb{Z} * l_2 \mapsto \mathbb{Z} \vdash \\ \quad \mathrm{let}\ x \Leftarrow \mathrm{read}_{l_1}\ \mathrm{in} \\ \quad \mathrm{let}\ y \Leftarrow \mathrm{read}_{l_2}\ \mathrm{in} \\ \quad \mathrm{let}\ \star \Leftarrow \mathrm{store}_{l_2}(x + y \leq 10)\ \mathrm{in} \\ \quad\quad \mathrm{val}_{l_1 \mapsto \mathbb{Z} * l_2 \mapsto \mathbb{B}}\ \star \\ : 1; l_1 \mapsto \mathbb{Z} * l_2 \mapsto B \end{array}$$

This program starts in states with two locations l_1 and l_2 containing integers, reads an integer from both of them and stores the boolean result of the test in l_2. Note that in the final state type of the program, the type of l_2 has changed to $\mathbb{B}$. Also note that we have not had to state all the context preserved by each of the basic operations on the state; this is due to the use of the symmetric monoidal lifting.

If we do not assume that the operation $\otimes$ on state types is commutative then we can use this calculus as an improved language for the session types of Sections 3.2.2 and 4.2.4. We restate the example program from Section 3.2.2, using the new let rule to ensure that sub-programs are oblivious to the whole program's state type:

$$\begin{array}{l} -; ?\mathit{Int}.?\mathit{Int}.(!\mathit{Int}.\circ + \circ) \vdash \\ \quad \text{let } x \Leftarrow \text{input}_{\mathit{Int}} \text{ in} \\ \quad \text{let } y \Leftarrow \text{input}_{\mathit{Int}} \text{ in} \\ \quad \text{let } z \Leftarrow \text{val } (x + y) \text{ in} \\ \quad\quad \text{if } z > 10 \text{ then output}_{\mathit{Int}}\ z \text{ else val } \star_1 \\ \quad : 1; \circ \end{array}$$

Notice that we have also been able to remove the explicit uses of the partial order on session types due to the C-Conseq rule.

5.2 Equational Theory and Interpretation

The equational rules for the Symmetric Monoidal Typed Command Calculus are presented in Figure 4, supplemented by axioms, reflexivity, symmetry, transitivity and congruence as usual. The rules for the value calculus are unchanged. The state calculus now has additional $\beta\eta$ rules for both of the type constructors. We use Ghani's generalised η rule (Ghani, 1995), which eliminates the need for commuting conversions.

The command calculus retains the inclusion of value and state equalities, the $\beta\eta$ rules for the unary sequencing construct and the β rule for functions from before. There are also $\beta\eta$ rules for the pair and unit sequencing constructs. There are also two β rules for the pair and unit sequencing constructs that cross the divide between eliminations of product and unit types performed in the command calculus and those performed in the state calculus. This is required to establish completeness. Finally, there are three sets of commuting conversion rules, one for each of the sequencing constructs.

As before the equational rules generate three equational judgments of the form $\Gamma \vdash^{\mathsf{v}} e_1 = e_2 : A$, $\Delta \vdash^{\mathsf{s}} s_1 = s_2 : S$ and $\Gamma; \Delta \vdash^{\mathsf{c}} c_1 = c_2 : A; S$. By extending the interpretation above, the equational theory generated is sound and complete for closed double parameterised Freyd categories. See (Atkey, 2006) for the proof.

Theorem 4

The Symmetric Monoidal Typed Command Calculus is sound and complete for closed double parameterised Freyd categories.

State Calculus:

$$\begin{array}{rcl}
\text{let } (z_1, z_2) = (s_1, s_2) \text{ in } s_3 & = & s_2[s_1/z_1, s_2/z_2] \\
\text{let } (z_1, z_2) = s_1 \text{ in } s_2[z_1 \otimes z_2/z] & = & s_2[s_1/z] \\
\text{let } \star_I = \star_I \text{ in } s_2 & = & s_2 \\
\text{let } \star_I = s_1 \text{ in } s_2[\star_I/z] & = & s_2[s_1/z]
\end{array}$$

Value calculus:

$$\begin{array}{rcl}
\pi_i(e_1, e_2) & = & e_i \\
e & = & (\pi_1 e, \pi_2 e) \\
e & = & \star \\
f & = & (\lambda(x^A; z^{S_1}).f(x; z))
\end{array}$$

Command Calculus:

$$\frac{e_1 = e_2 \qquad s_1 = s_2}{(e_1; s_1) = (e_2; s_2)}$$

$$\begin{array}{rcl}
\text{let } (x; z) \Leftarrow (e; s) \text{ in } c & = & c[e/x, s/z] \\
\text{let } (x; z) \Leftarrow c \text{ in } (x; z) & = & c \\
\text{let } (x; z_1, z_2) \Leftarrow (e; (s_1, s_2)) \text{ in } c & = & c[e/x, s_1/z_1, s_2/z_2] \\
\text{let } (x; z_1, z_2) \Leftarrow c \text{ in } (x; (z_1, z_2)) & = & c \\
\text{let } (x; \star_I) \Leftarrow (e; \star_I) \text{ in } c & = & c[e/x] \\
\text{let } (x; \star_I) \Leftarrow c \text{ in } (x; \star_I) & = & c \\
(\lambda(x, z).c)\ (e, z') & = & \text{let } (x; z) \Leftarrow (e; z') \text{ in } c \\
\text{let } (x; z_1, z_2) \Leftarrow (e_1; s_1) \text{ in } (e_2; s_2) & = & (e_2[e_1/x]; \text{let } (z_1, z_2) = s_1 \text{ in } s_2) \\
\text{let } (x; \star_I) \Leftarrow (e_1; s_1) \text{ in } (e_2; s_2) & = & (e_2[e_1/x]; \text{let } \star_I = s_1 \text{ in } s_2) \\
C[\text{let } (x; z) \Leftarrow c_1 \text{ in } c_2] & = & \text{let } (x; z) \Leftarrow c_1 \text{ in } C[c_2] \\
C[\text{let } (x; z_1, z_2) \Leftarrow c_1 \text{ in } c_2] & = & \text{let } (x; z_1, z_2) \Leftarrow c_1 \text{ in } C[c_2] \\
C[\text{let } (x; \star_I) \Leftarrow c_1 \text{ in } c_2] & = & \text{let } (x; \star_I) \Leftarrow c_1 \text{ in } C[c_2]
\end{array}$$

$$C[-] ::= - \mid \text{let } (x; z) \Leftarrow C[-] \text{ in } c \mid \text{let } (x; z_1, z_2) \Leftarrow C[-] \text{ in } c \mid \text{let } (x; \star_I) \Leftarrow C[-] \text{ in } c$$

Fig. 4. Equational Rules for the Monoidal Typed Command Calculus

6 Related Work

Computational monads (Moggi, 1989; Moggi, 1991) have been extremely successful in providing a framework for modelling a large range of computational phenomena. They have also be used to do effectful programming in pure functional languages (Peyton-Jones & Wadler, 1993). Power and Robinson introduced Freyd Categories (Power & Robinson, 1997) as an alternative presentation of strong monads.

In this paper we have presented the basic category theoretic definitions for interpreting type systems with additional information about the effects that programs perform. We have given typed calculi that directly correspond to these definitions. In this section we cite some of the previous work on such type systems and relate them to the present work.

6.1 Linear Types

The problem of incorporating state and side-effects into functional languages has been attacked by using type systems based on variants of Girard's Linear Logic (Girard, 1987). Examples include Wadler's systems (Wadler, 1990; Wadler, 1991), Hofmann's LFPL (Hofmann, 2000) and Morrisett *et al*'s Linear Language with Locations (Morrisett *et al.*, 2005). The last of these also uses indexed types to separate pointers from assertions about their use. See also Walker's chapter (Walker, 2005).

In (Atkey, 2006) we demonstrated how to use our parameterised notions of computation to interpret a language with linear types. We take Hofmann's LFPL as a prototypical linearly typed language; this language is similar to those of Wadler (Wadler, 1990) and Walker (Walker, 2005) . The language LFPL is designed so that every data structure stored in the heap has a single pointer to it, so that when it is used its heap space may be safely made available back to the program. The key point in LFPL's type system (and most other linear systems) is that references to the heap must be treated linearly (no duplication or discarding) in order to preserve the single-pointer invariant. A subset of types in the language are labelled as heap-free: that is, they do not refer to the memory of the computer. Hence they may be treated non-linearly.

We use double parameterised Freyd categories to model this situation. The types of the calculus are modelled as objects in $\mathcal{K}$, i.e. as pairs of $\mathcal{C}$ and $\mathcal{S}$ objects. A heap-free type has an $\mathcal{S}$ component which is just I, the unit of the monoidal structure of $\mathcal{S}$. It may then be freely duplicated and discarded. We also require the combined premonoidal structures on $\mathcal{K}$ to be symmetric monoidal. This amounts to the following diagram commuting:

$$\begin{tikzcd}
(A \times B, S_1 \otimes S_2) \arrow[r, "(c_1 \otimes_{\mathcal{C}} B) \otimes_{\mathcal{S}} S_2"] \arrow[d, "A \otimes_{\mathcal{C}} (S_1 \otimes_{\mathcal{S}} c_2)"'] & (A' \times B, S_1' \otimes S_2) \arrow[d, "A' \otimes_{\mathcal{C}} (S_1' \otimes_{\mathcal{S}} c_2)"] \\
(A \times B', S_1 \otimes S_2') \arrow[r, "(c_1 \otimes_{\mathcal{C}} B') \otimes_{\mathcal{S}} S_2'"] & (A' \times B', S_1' \otimes S_2')
\end{tikzcd}$$

for all $c_1 : (A, S_1) \to (A', S_1')$ and $c_2 : (B, S_2) \to (B', S_2')$. In terms of parameterised monads this is the requirement that the two obvious arrows of type

$$T(S_1, S_1', A) \times T(S_2, S_2', B) \to T(S_1 \otimes S_2, S_1' \otimes S_2', A \times B)$$

using strength and lifting are equal. We also require that the functor $J(-, I)$ is full – meaning that no effects may occur with the empty state description.

From this structure we can derive a functor $J' : \mathcal{C} \to \mathcal{K}$, defined as $J'(-) = J(-, I)$, which is full and preserves finite products. The category $\mathcal{K}$ is symmetric monoidal with $(A, S_1) \otimes (B, S_2) = (A \times B, S_1 \otimes S_2)$. If the original double parameterised Freyd category had exponentials then this induces an adjunction:

$$\mathcal{K}(J'\Gamma \otimes X, Y) \cong \mathcal{C}(\Gamma, X \Rightarrow Y)$$

suitable for interpreting functions that do not have any free non-heap-free variables. In order to interpret functions that close over non-heap-free variables we need a second closed structure which will make $\mathcal{K}$ a symmetric monoidal *closed* category. At the level of double parameterised Freyd categories this requires a second adjoint pair:

$$\mathcal{K}((A \times B, S_1 \otimes S_2), (C, S_3)) \cong \mathcal{K}((A, S_1), (B, S_2) \multimap (C, S_3)).$$

This kind of function allows closure over state which is hidden from clients of the closure. Notice that the codomain on the right hand side is a single object, rather than a pair of a $\mathcal{C}$ and a $\mathcal{S}$ object. Due to the definition of parameterised Freyd category this must actually be such a pair. In (Atkey, 2006) we considered an example using functor categories to model a linear language with state. In this case the $\mathcal{C}$ component is just the terminal object.

Clean's uniqueness types (Barendsen & Smetsers, 1993) use a linear discipline to incorporate effects into a pure functional language, but the approach is too different to other linearly typed languages to fit into the method described in this section. Harrington gives a proof theory and categorical semantics for uniqueness types (Harrington, 2006).

6.2 Indexed Types

The idea of annotating typing judgments with start and finish annotations about the state of the machine has appeared in several type systems in the literature. Alias Types (Smith *et al.*, 2000; Walker & Morrisett, 2000), the Calculus of Capabilities (Walker *et al.*, 2000), Hoare Type Theory (Nanevski *et al.*, 2006) and Applied Type Systems with Stateful Views (Zhu & Xi, 2005) all define an additional syntactic category of state descriptions to safely type pointer manipulating programs. The primary difference with our work is that they all index type judgments by contexts of pointer values, enabling them to divorce pointers and assertions that they may be accessed. This is particularly vital in the example of typed state with *read* and *store* operations annotated with explicit locations in Sections 4.2.2 and 5.1.2, since it allows functions that are parametric in the locations they operate on to be written.

We briefly sketch the additions to the Typed Command Calculus with implicit state described in Section 5.1.2. We extend the judgments with an additional context Θ which is a list of abstract location variables. Value and state types may now contain references to the location variables in the context, so we have a value type $\mathit{Ref}(l)$ of references to location l and the locations in state types are bound by Θ.

In this language, the *read* and *store* operations are typed as follows:

$$\frac{\Theta \mid \Gamma \vdash^{\mathsf{v}} x : \mathit{Ref}(l)}{\Theta \mid \Gamma; l \mapsto X \vdash^{\mathsf{c}} \mathrm{read}\ x : X; l \mapsto X}$$

$$\frac{\Theta \mid \Gamma \vdash^{\mathsf{v}} x : \mathit{Ref}(l) \qquad \Theta \mid \Gamma \vdash^{\mathsf{v}} y : X}{\Theta \mid \Gamma; l \mapsto ? \vdash^{\mathsf{c}} \mathrm{store}\ x\ y : 1; l \mapsto X}$$

Hence the dynamic location x for reading and storing is determined at runtime. For typing, the location is statically fixed by l, but l is now a location variable rather than a fixed location as in Section 4.2.1. Note that the type X here could also contain references of type $\mathit{Ref}(l')$ for some other location variable l', allowing linked data structures on the heap to be represented. More complex forms of state descriptions would allow more complex linked data structures to be considered, but consideration of such is beyond the scope of this paper.

The rules for function types become more complicated:

$$\frac{\Theta, \Theta' \mid \Gamma, x : A; S_1 \vdash^{\mathsf{c}} c : B; S_2 \qquad \Theta \vdash \Gamma}{\Theta \mid \Gamma \vdash^{\mathsf{v}} \lambda\Theta'; x : A; S_1.c : \Pi\Theta'.(A; S_1) \to (B; S_2)}$$

$$\frac{\Theta \mid \Gamma \vdash^{\mathsf{v}} e_1 : \Pi\Theta'.(A; S_1) \to (B; S_2) \qquad \overrightarrow{\Theta \vdash l} \qquad \Theta \mid \Gamma \vdash^{\mathsf{v}} e_2 : A[\overrightarrow{l}/\Theta']}{\Theta \mid \Gamma; S_1[\overrightarrow{l}/\Theta'] \vdash^{\mathsf{c}} e_1[\overrightarrow{l}]e_2 : B[\overrightarrow{l}/\Theta']; S_2[\overrightarrow{l}/\Theta']}$$

Here, the judgment $\Theta \vdash \Gamma$ means that Γ is well-formed with respect to the abstract location variables in Θ. The function introduction rule abstracts over a context Θ' of abstract location variables, a value variable x and a state type S_1. Function application takes a list of abstract locations $\overrightarrow{l}$ to be substituted into e_1's type for the variables in Θ'.

To interpret such a system we can use an indexed parameterised Freyd category. That is, we have a category $\mathcal{I}$ for interpreting contexts of abstract location variables and a functor from $\mathcal{I}^{\mathsf{op}}$ to the category of parameterised Freyd categories. The rules of the calculus are then interpreted as standard in indexed and dependently typed systems (Taylor, 1999).

There is nothing special about side-effects in the above example. Using the framework of parameterised Freyd categories we may easily alter the above type system to cope with indexed session types or multiple I/O devices. Moreover, using the construction sketched in the previous section to derive a model of a linear type system from special double parameterised Freyd categories to get a symmetric monoidal closed category with a full subcategory with finite products can be replayed in this setting. We conjecture that such an indexed structure can be used to interpret a language similar to L^3 (Morrisett *et al.*, 2005).

Most of the work on indexed types above has been presented using operational semantics. An exception is the work on Separation-Logic typing by Birkedal, Torp-Smith and Yang (Birkedal *et al.*, 2006). They describe a type system for Idealised Algol based on the assertions of Separation Logic. They refine the type `comm` to types of the form $\{P\}$–$\{Q\}$, where P and Q are assertions about the start and end state. Their model uses functors $\mathsf{tri}' : \mathcal{P}^{\mathsf{op}} \times \mathcal{P} \to \mathcal{D}$, where $\mathcal{P}$ is the category of assertions and $\mathcal{D}$ is their category of program interpretations to interpret these types, along with a sequencing operation that appears to be similar to our definition of parameterised monad. Differences arise due to the active nature of types in call-by-name Idealised Algol compared to the passive values in the call-by-value languages we have considered.

6.3 Composable Continuations

We have already mentioned Wadler's work on expressing composable continuations in terms of monads (Wadler, 1994). He came close to the definition we have presented here for parameterised monad, but pointed out that it was not a monad. We have presented a justification for parameterised monads by showing their relationship with parameterised adjunctions and by presenting several examples.

6.4 Type and Effect Systems

Effect Systems (Lucassen & Gifford, 1988) augment traditional type systems with information about the side-effects caused by a program's execution. Wadler (Wadler & Thiemann, 2003) has presented a connection between effect systems and monads indexed by effect types. In concurrent work with this paper we have investigated using parameterised monads to interpret a type and effect system for reading and writing. The basic idea is to consider a state category with objects that are members of the power set of $\{\mathsf{r}(l), \mathsf{w}(l) \mid l \in L\}$ for some set of locations L. The types of the read and store operations then become $\mathit{read}_l : 1 \to T(\{\mathsf{r}(l)\}, \{\mathsf{r}(l)\}, V)$ and $\mathit{store}_l : V \to T(\{\mathsf{w}(l)\}, \{\mathsf{w}(l)\}, 1)$. The intuitive notion here is that the objects of the state category represent sets of permissions: e.g. $\mathsf{r}(l)$ represents the permission to read location l. The lifting of the operation of set union on sets of permissions, in the same manner as symmetric monoidal lifting in this paper, is essential in order to type realistic programs. The tricky part comes in defining the parameterised monad for $T(S_1, S_2, A)$, where S_1 and S_2 are sets of permissions. Benton *et al* (Benton *et al.*, 2006) do this by considering the relations that pure reading and pure writing state transformations preserve. In work concurrent with this paper, we have taken a more intensional approach and considered a variation on Plotkin and Power's algebraic presentation of computational monads for parameterised monads.

7 Conclusions

We have presented generalisations of Moggi's computational monads and Power *et al*'s Freyd categories to cover parameterised effects, our main examples being typed side-effects and various forms of typed I/O. By also considering monoidal parameterisation, our definitions also cover separated side-effects, multiple streams of I/O, simple session types and effect types. We have also presented two typed λ-calculi which are sound and complete for the simple parameterisation and symmetric monoidal parameterisation cases.

We have also discussed the relationship between our semantic definitions and existing type systems for effects present in the literature. In the case of linear types this involves the imposition of additional constraints on our definitions to get a symmetric monoidal category. An unresolved aspect of this is a nice account of closure over linearly typed variables, thus capturing some state in the function. We have also briefly discussed the relationship between our non-indexed calculi and the indexed calculi present in the literature. A point for future work here is to create a semantics for typed state that allows polymorphism over state descriptions so that state shared by several functions may be hidden from the rest of the program; we expect that this problem is related to the problem of interpreting linear function types.

Finally, Plotkin and Power's approach of deriving computational monads from algebras of operations and equations (Plotkin & Power, 2002) should be adaptable to parameterised monads. We have already done a small amount of work in this direction by deriving the global typed state monad above from a plausible algebra of lookup and update operations (see the appendix of (Atkey, 2006)). We have also done some work in treating a type and effect system for reading and writing in the framework of algebras for parameterised monads.

Acknowledgements This work was funded by the ReQueST grant (EP/C537068) from the Engineering and Physical Sciences Research Council. Thanks to the anonymous MSFP and JFP reviewers for helpful comments and spotting of various technical errors.

References

Atkey, Robert. (2006). *Substructural Simple Type Theories for Separation and In-place Update*. Ph.D. thesis, University of Edinburgh.

Barendsen, Erik, & Smetsers, Sjaak. (1993). Conventional and Uniqueness Typing in Graph Rewrite Systems (extended abstract). *Pages 41–51 of:* Shyamasundar, R. K. (ed), *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science*. LNCS, vol. 761. Springer.

Benton, Nick, Kennedy, Andrew, Hofmann, Martin, & Beringer, Lennart. (2006). Reading, Writing and Relations. *Pages 114–130 of: Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*. LNCS, vol. 4279. Springer.

Birkedal, Lars, Torp-Smith, Noah, & Yang, Hongseok. (2006). Semantics of Separation-Logic Typing and Higher-order Frame Rules for Algol-like Languages. *Logical Methods in Computer Science*, **2**(5).

Danvy, Olivier, & Filinski, Andrzej. (1989). *A Functional Abstraction of Typed Contexts*. Tech. rept. 89/12. DIKU – Computer Science Department, University of Copenhagen.

Ghani, Neil. (1995). *Adjoint Rewriting*. Ph.D. thesis, University of Edinburgh.

Girard, Jean-Yves. (1987). Linear Logic. *Theoretical Computer Science*, **50**, 1–101.

Harrington, Dana. (2006). Uniqueness Logic. *Theoretical Computer Science*, **354**(1), 24–41.

Hofmann, Martin. (2000). A Type System for Bounded Space and Functional In-Place Update. *Nordic Journal of Computing*, **7**(4), 258–289.

Levy, Paul Blain, Power, John, & Thielecke, Hayo. (2003). Modelling environments in call-by-value programming languages. *Information and Computation*, **185**(2), 182–210.

Lucassen, J. M., & Gifford, D. K. (1988). Polymorphic effect systems. *Pages 47–57 of: POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM.

Mac Lane, Saunders. (1998). *Categories for the Working Mathematician*. 2nd edn. Graduate Texts in Mathematics, no. 5. Springer.

Moggi, Eugenio. (1989). Computational lambda-calculus and monads. *Pages 14–23 of:*

Parikh, Rohit (ed), *Proceedings of the Fourth Annual IEEE Symp. on Logic in Computer Science, LICS 1989*. IEEE Computer Society Press.

Moggi, Eugenio. (1991). Notions of Computation and Monads. *Information and Computation*, **93**(1), 55–92.

Morrisett, Greg, Ahmed, Amal J., & Fluet, Matthew. (2005). L^3: A Linear Language with Locations. *Pages 293–307 of: Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*. LNCS, vol. 3461. Springer.

Nanevski, A., Morrisett, G., & Birkedal, L. (2006). Polymorphism and Separation in Hoare Type Theory. *Pages 62–73 of:* Reppy, John H., & Lawall, Julia L. (eds), *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*. ACM.

Nielson, Flemming, & Nielson, Hanne Riis. (1996). From CML to its process algebra. *Theoretical Computer Science*, **155**(1), 179–219.

O'Hearn, Peter, Reynolds, John, & Yang, Hongseok. (2001). Local Reasoning about Programs that Alter Data Structures. *Pages 1–19 of:* Fribourg, Laurent (ed), *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. LNCS, vol. 2142. Springer.

O'Hearn, Peter W., Yang, Hongseok, & Reynolds, John C. (2004). Separation and information hiding. *Pages 268–280 of:* Jones, Neil D., & Leroy, Xavier (eds), *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. ACM.

Peyton-Jones, Simon L., & Wadler, Philip. (1993). Imperative functional programming. *Pages 71–84 of: POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM.

Plotkin, Gordon D., & Power, John. (2002). Notions of Computation Determine Monads. Nielsen, M., & Engberg, U. (eds), *Foundations of Software Science and Computation Structures : 5th International Conference, FOSSACS 2002*. LNCS, vol. 2303. Springer.

Power, John, & Robinson, Edmund. (1997). Premonoidal Categories and Notions of Computation. *Math. Struct. in Comp. Science*, **7**(5), 453–468.

Power, John, & Thielecke, Hayo. (1999). Closed Freyd- and kappa-categories. *Pages 625–634 of: Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings*. LNCS, vol. 1644. Springer.

Skalka, Christian, & Smith, Scott. (2004). History Effects and Verification. *Pages 107–128 of:* Chin, Wei-Ngan (ed), *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*. LNCS, vol. 3302. Springer.

Smith, Frederick, Walker, David, & Morrisett, J. Gregory. (2000). Alias types. *Pages 366–381 of: Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*. LNCS, vol. 1782. Springer.

Taylor, Paul. (1999). *Practical Foundations of Mathematics*. Cambridge studies in advanced mathematics, no. 59. Cambridge University Press.

Thielecke, Hayo. (1997). *Categorical Structure of Continuation Passing Style*. Ph.D. thesis, University of Edinburgh.

Uustalu, T. (2003). Generalizing substitution. *Theor. Inform. and Appl.*, **37**(4), 315–336.

Vasconcelos, Vasco T., Gay, Simon J., & Ravara, Antonió. (2006). Typechecking a Multithreaded Functional Language with Session Types. *Theoretical Computer Science*, **368**(1–2), 64–87.

Wadler, Philip. (1990). Linear types can change the world! Broy, M., & Jones, C. (eds), *Programming Concepts and Methods*. North Holland, Amsterdam.

Wadler, Philip. (1991). Is there a use for linear logic? *Pages 255–273 of: PEPM '91: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. New York, NY, USA: ACM.

Wadler, Philip. (1994). Monads and composable continuations. *Lisp and Symbolic Computation*, **7**(1), 39–56.

Wadler, Philip, & Thiemann, Peter. (2003). The marriage of effects and monads. *ACM Trans. Comput. Log.*, **4**(1), 1–32.

Walker, David. (2005). Substructural Type Systems. *Pages 3–43 of:* Pierce, Benjamin C. (ed), *Advanced Topics in Types and Programming Languages*. MIT Press.

Walker, David, & Morrisett, J. Gregory. (2000). Alias types for recursive data structures. *Pages 177–206 of:* Harper, Robert (ed), *Types in Compilation, Third International Workshop, TIC 2000, Montreal, Canada, September 21, 2000, Revised Selected Papers*. LNCS, vol. 2071. Springer.

Walker, David, Crary, Karl, & Morrisett, Greg. (2000). Typed Memory Management via Static Capabilities. *ACM Trans. Program. Lang. Syst.*, **22**(4), 701–771.

Zhu, Dengping, & Xi, Hongwei. (2005). Safe Programming with Pointers through Stateful Views. *Pages 83–97 of: Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*. LNCS, vol. 3350. Springer.