Parameterised Notions of Computation

Robert Atkey
LFCS, School of Informatics
University of Edinburgh, Mayfield Road
Edinburgh EH9 3JZ, UK
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 such as exceptions, side-effects, input/output and continuations. We present generalisations of both constructs, 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 monoidal 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 monoidal parameterisation — and act as prototypical languages with parameterised effects.

Keywords: Freyd Categories, Computational Monads, Computational Effects

1. INTRODUCTION

Moggi's framework of Computational Monads [9, 8], and Power *et al*'s equivalent notion of Freyd Categories [12, 13, 5], have been successfully used to capture a wide range of computational effects used in programming language designs, such as non-termination, 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 main motivating example is that of side-effects. The standard side-effects monad selects an object S of the base category to represent the type of the computer's store and sets the functor part of the monad to be $TA = (S \times A)^S$. Computations 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 covers all the possible stores that a program can generate. For the purposes of modelling features such as strong update [10], where the type of storage cells may change over time, or type systems inspired by Hoare Logic such as Alias Types [15], this is inadequate. Such type systems type the current store explicitly and restrict the range of possible operations according to the current type. We will present two type systems with explicitly typed stores below in Sections 3 and 5. An example judgement has the form:

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

The context Γ and type A are the traditional value context and result type respectively. The context $z : S_1$ and type S_2 use the types S_1 and S_2 to 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 be composed of functors $T : \mathcal{S}^{\mathrm{op}} \times \mathcal{S} \times \mathcal{C} \to \mathcal{C}$, with additional conditions on the unit and multiplication that we set out in Section 2.1. In the case of state we assume a functor $\widehat{\cdot} : \mathcal{S} \to \mathcal{C}$ and set $T(S_1, S_2, A) = (\widehat{S_2} \times A)^{\widehat{S_1}}$.

This situation suffices for modelling explicitly typed global state. Below, we present two other examples, 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 [2].

In many cases, however, the assumption of globalness is too strong. We can regard the store of a computer as being built from multiple independent regions, right down to the individually addressable storage cells. Similarly, the computer may have multiple I/O devices attached and be able to send output and receive input from them independently.

In order to model this situation, we assume that the parameterising category $\mathcal{S}$ is symmetric monoidal. That is, there is 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.

In Section 2, we present our definitions of parameterised monad and parameterised Freyd category, prove them equivalent and give our main examples: typed side-effects, 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 symmetric monoidal parameterisation, extending the range of examples to allow separated side-effects and multiple streams of input/output. We extend the Typed Command Calculus to this situation in Section 5. Finally, in Sections 6 and 7, we describe related work and present some concluding remarks and future work.

2. SIMPLE PARAMETERISATION

Simple parameterisation allows us to model situations where the current state is global; e.g. a store that cannot be subdivided, or a single I/O port, or a single continuation context. We first give the definition for parameterised strong monads, and then some examples demonstrating the use of the parameterisation. We then define parameterised Freyd categories and show that the two are equivalent. In the next section we will describe a typed λ-calculus is sound and complete for parameterised Freyd categories (and parameterised strong monads). All of our definitions are over a base category $\mathcal{C}$, which we assume to have finite products, and a parameterising category $\mathcal{S}$. We refer to these as the *value* and *state* categories respectively.

2.1. Parameterised Strong Monads

Recall that a monad consists of a functor $T : \mathcal{C} \to \mathcal{C}$ and two natural transformations, the unit $\eta_A : A \to TA$ and the multiplication $\mu_A : T(TA) \to TA$. For modelling effects, arrows $A \to TB$ are regarded as computations that take a value of type A, do some computation and produce a value of type B. The unit is used to represent the identity computation and the multiplication is used to interpret the sequencing of computations; i.e. a computation that produces a computation that produces a value is sequenced to give a computation that produces a value.

As described in the introduction, for *parameterised* monads we extend the functor part to be a functor $T : \mathcal{S}^{\mathsf{op}} \times \mathcal{S} \times \mathcal{C} \to \mathcal{C}$. On objects, the first and second arguments represent the assertions about the pre-state and post-state respectively. On arrows, they represent the strengthening of pre-conditions and the weakening of post-conditions, hence the contravariance in the first argument and covariance in the second.

We extend the unit of the monad to be a family of arrows $\eta_{S,A} : A \to T(S,S,A)$, used for representing the identity computation at any state. This family of arrows is required to be natural in A, as for non-parameterised monads. It is also required to be dinatural ([7] §IX.4) in S, so the equation $\eta_{S_1,A}; T(S_1,f,A) = \eta_{S_2,A}; T(f,S_2,A)$ must hold for any $f : S_1 \to S_2$. The intuitive reading of dinaturality in this case is that 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 $\mu_{S_1,S_2,S_3,A} : T(S_1,S_2,T(S_2,S_3,A)) \to T(S_1,S_3,A)$. Only pairs of computations where the former's post-condition matches the latter's pre-condition 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 $f : S_2 \to S_2'$:

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

This states that if we have two commands with a mismatch in the intermediate state, then it does not matter if we weaken the former's post-condition, or the latter's pre-condition in order to make them match.

Definition 1 *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 transformation $\eta_{S,A} : A \to T(S,S,A)$, natural in A and dinatural in S; and a transformation $\mu_{S_1,S_2,S_3,A} : T(S_1,S_2,T(S_2,S_3,A)) \to T(S_1,S_3,A)$, natural in S_1, S_3 and A and dinatural in S_2. These must obey the monad laws: $\eta;\mu = T(S_1,S_2,\eta);\mu = id$ and $T(S_1,S_2,\mu);\mu = \mu;\mu$.*

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 $\mathcal{S}$. Therefore, 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.

Crucial to Moggi's use of monads to model effectful computation is the notion of strength. A strength for a monad is a natural transformation $\tau_{A,B} : A \times TB \to T(A \times B)$, obeying some axioms. Strength is used to interpret computations in context: given a computation represented as an arrow $c : A \to TA'$ and a context B, we can lift c up to the larger context by composition with the strength: $B \times A \xrightarrow{B \times c} B \times TA' \xrightarrow{\tau} T(B \times A')$. The idea of computation in context forms the main focus of the definition of premonoidal structure, which is a major part of the definition of Freyd category (see below). We also extend the computation in context idea in Section 4 below to cover computation in state context.

The definition of strength generalises easily to parameterised monads:

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

Definition 2 *A* strength *for a parameterised monad* (T, η, μ) *is a natural transformation* $\tau_{A,S_1,S_2,B} : A \times T(S_1, S_2, B) \to T(S_1, S_2, A \otimes B)$ *that obeys the obvious adaptations of the axioms for non-parameterised strength.*

2.2. Examples

We now give some examples of parameterised monads modelling computational effects that require the additional parameterising category $\mathcal{S}$. In addition to these examples, note that any strong monad gives a strong parameterised monad with any parameterising category.

2.2.1. Typed State

As stated in the introduction, we can use parameterised monads to model typed global state. Select a collection of storable objects of $\mathcal{C}$, called $\mathcal{C}_0$. Set the category $\mathcal{S}$ to have objects $[A]$ for each object A of $\mathcal{C}_0$, plus an additional object $\Diamond$. The only non-identity arrows are $clear_A : [A] \to \Diamond$ for each $A \in \mathcal{C}_0$. Hence $\Diamond$ is the terminal object of $\mathcal{S}$.

The functor $\widehat{\cdot} : \mathcal{S} \to \mathcal{C}$ maps each $[A]$ to A and $\Diamond$ to the terminal object. Hence the objects $[A]$ are used to represent storage cells containing values of type A and $\Diamond$ represents a memory cell containing a value of indeterminate type.

The monad is defined as $T(S_1, S_2, A) = (\widehat{S_2} \times A)^{\widehat{S_1}}$. The definitions of the unit, multiplication and strength are almost identical to those for the standard side-effects monad. There are two basic operations for this monad, $read$ and $store$, which we define as follows:

$$\begin{array}{lcl@{\qquad}lcl} read & : & T([A], [A], A) & store & : & A \to T(\Diamond, [A], 1) \\ read & = & \lambda s.(s, s) & store & = & a \mapsto \lambda s.(a, \star) \end{array}$$

2.2.2. Typed I/O

For simplicity, we restrict to $\mathcal{C} = \mathrm{Set}$, the category of sets and functions, and $\mathcal{S}$ being a small discrete category. Assume a collection of input sets $\{I(S)\}_{S \in \mathcal{S}}$ and a collection of output sets $\{O(S_1, S_2)\}_{S_1, S_2 \in \mathcal{S}}$. The objects of $\mathcal{S}$ represent the states that an I/O device may be in; the sets $I(S)$ represent the possible inputs in a state; and $O(S_1, S_2)$ represent the possible outputs in a given state, and the states they cause the device to go into. On objects, the monad functor is built inductively from the following rules:

$$\frac{a \in A}{\mathsf{e}(a) \in T(S, S, A)} \qquad \frac{f \in I(S_1) \Rightarrow T(S_1, S_2, A)}{\mathsf{i}(f) \in T(S_1, S_2, A)} \qquad \frac{m \in O(S_1, S_2) \qquad c \in T(S_2, S_3, A)}{\mathsf{o}(m, c) \in T(S_1, S_3, A)}$$

Computations are therefore trees of output transitions and branching input transitions, terminated by result values. The monad unit just maps a to $\mathsf{e}(a)$ and multiplication concatenates trees.

The monad has two primitive operations that perform input and output:

$$\begin{array}{lcl@{\qquad}lcl} input & : & T(S, S, I(S)) & output & : & O(S_1, S_2) \to T(S_1, S_2, 1) \\ input & = & \mathsf{i}(\lambda x.\mathsf{e}(x)) & output & = & \lambda m.\mathsf{o}(m, \mathsf{e}(*)) \end{array}$$

2.2.3. Composable Continuations

Parameterised monads provide a way to interpret Danvy and Filinski's composable continuations [2]. 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 attempt to express composable continuations in terms of monads [18].

We require $\mathcal{C}$ to be cartesian closed, and set $\mathcal{S}$ to be $|\mathcal{C}|$, the discrete category with the same objects as $\mathcal{C}$. Define $T(R_1, R_2, A) = (A \to R_2) \to R_1$, where $\to$ is the functor part of the cartesian closed structure. Unit, multiplication and strength are defined as for the standard continuations monad [9].

In terms of the type system given by Danvy and Filinski in [2], a judgement $\rho, \alpha \vdash E : \tau, \beta$ is interpreted as an arrow $[\![\rho]\!] \to T([\![\beta]\!], [\![\alpha]\!], [\![\tau]\!])$. The $reset$ operator is interpreted as an arrow in $\mathcal{C}$, using the internal language of $\mathcal{C}$:

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

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

$$\begin{array}{rcl} shift & : & ((A \to T(C, C, B)) \to T(E, D, D)) \to T(E, B, A) \\ shift & = & f \mapsto \lambda k.(e(\lambda v.\lambda k'.k'(kv)))(\lambda x.x) \end{array}$$

See below and [2] and [18] 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 [17].

2.3. 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 pair of functors $A \ltimes - : \mathcal{K} \to \mathcal{K}$ and $- \rtimes A : \mathcal{K} \to \mathcal{K}$ that agree on objects: $A \ltimes B = A \rtimes B = A \otimes B$, 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, $A \ltimes f; g \rtimes B' = g \rtimes B; A' \ltimes f$. 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 is this way for two reasons. Firstly, we want the objects of the codomain category to be comprised to 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 obvious 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 $\mathcal{K}$ and saying that J should preserve them.

Definition 3 *A* parameterised Freyd category *consists of three categories $\mathcal{C}$, $\mathcal{S}$ and $\mathcal{K}$, where $\mathcal{C}$ has finite products, and three functors $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$, $\ltimes_{\mathcal{C}} : \mathcal{C} \times \mathcal{K} \to \mathcal{K}$ and $\rtimes_{\mathcal{C}} : \mathcal{K} \times \mathcal{C} \to \mathcal{K}$, such that:*

1. *J is identity on objects;*
2. *The monoidal structure of $\mathcal{C}$ is respected: $A \ltimes_{\mathcal{C}} J(B, X) = J(A, X) \rtimes_{\mathcal{C}} B = J(A \times B, X)$ and $f \ltimes_{\mathcal{C}} J(g, s) = J(f, s) \rtimes_{\mathcal{C}} g = J(f \times g, s)$;*
3. *For each $S \in$ Ob$\mathcal{S}$, the transformations $J(\alpha_{ABC}, S)$, $J(\lambda_A, S)$, $J(\rho_A, S)$ and $J(\sigma_{A,B}, S)$ must be natural in all combinations of $\otimes, \ltimes_{\mathcal{C}}$ and $\rtimes_{\mathcal{C}}$ that make up their domain and codomain.*

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 $\ltimes_{\mathcal{C}}$ and $\rtimes_{\mathcal{C}}$. Closure for parameterised Freyd categories is similar to that for Freyd categories. It will be used to interpret function types.

Definition 4 *A parameterised Freyd category* $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$ *is* closed *if, for all* $A \in \mathrm{Ob}\mathcal{C}$ *and* $S \in \mathrm{Ob}\mathcal{S}$*, the functor* $J(- \times A, S) : \mathcal{C} \to \mathcal{K}$ *has a specified right adjoint, written* $(A, S) \to - : \mathcal{K} \to \mathcal{C}$*.*

We now show the relationship between parameterised Freyd categories and strong parameterised monads. To do this we shall go through parameterised adjunctions. The definition of parameterised adjunction is as follows:

Definition 5 *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:*

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

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

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

By Theorem §IV.7.3 in [7], 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 $\mathcal{S}^{\mathrm{op}} \times \mathcal{D} \to \mathcal{C}$ such that it is a parameterised adjunction in the sense of this definition. Using this result, 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 $(1, S) \to -$.

Now, 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 and parameterised monad, as for monads. See the appendix of [1] for more details.

Lemma 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* $\mathcal{C}$*, define a category* $\mathcal{C}_T$ *with objects pairs of* $\mathcal{C}$ *and* $\mathcal{S}$ *objects and* $\mathcal{C}_T((A_1, S_1), (A_2, S_2)) = \mathcal{C}(A_1, T(S_1, S_2, A_2))$*. The functors* $F = S, A \mapsto (A, S); s, f \mapsto \eta; T(S_1, s, f) : \mathcal{S} \times \mathcal{C} \to \mathcal{C}_T$ *and* $G = S_1, (A, S_2) \mapsto T(S_1, S_2, A); s, c \mapsto T(s, S_2, c); \mu : \mathcal{S} \times \mathcal{C}_T \to \mathcal{C}$ *form a parameterised adjoint pair.*

Proof Almost identical to the proof of the definition of a monad from an adjunction [7]. 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 [12], which links the premonoidal structure of Freyd categories with monad strength, to the parameterised case:

Lemma 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 operations are inverse.*

Proof Given a strength τ, define $f \otimes_{\mathcal{C}} c = f \otimes c; \tau_{A,S_1,S_2,B}$, $\otimes_{\mathcal{C}}$ 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 $\mathcal{C}_T$. □

Lemma 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 closure functors are identical in both cases. □

These lemmas combine to give:

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

3. TYPED COMMAND CALCULUS

In this section we define a typed λ-calculus, 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 [5].

3.1. Typing Rules

Levy *et al*'s fine-grain call-by-value calculus has two typing judgements $\Gamma \vdash^{\mathsf{v}} V : A$ and $\Gamma \vdash^{\mathsf{p}} M : A$. The first is used to type values, i.e. computations with no effect, and is interpreted in $\mathcal{C}$. The second judgement is used to type producers, i.e. computations with effects, and is interpreted in $\mathcal{K}$. The two main constructs of the calculus are typed as follows:

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

The construct $\texttt{produce}\ V$ incorporates values into producers and is interpreted using the functor J of a Freyd category. 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. This is interpreted using the premonoidal structure of the Freyd category and composition.

The Typed Command Calculus also has a typing judgement for each category present in the definition of parameterised Freyd category. For the three categories, there are three judgements:

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

The first is used to type values, and is 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 second is used to type state manipulations, and is interpreted in the state category $\mathcal{S}$. State manipulation terms consist of variables and primitive functions. The third judgement form is used to type computations, and is interpreted in the category $\mathcal{K}$. Computation terms are comprised of pure value and state terms, sequencing, primitive computations and function application. Contexts for computation judgements are a pair of a value context and a state context, and result types are a pair of a value typed and a state type. The typing rules for these judgements are given in Figure 1.

Let $\mathcal{T}_V$ and $\mathcal{T}_S$ be sets of primitive value types and primitive state types. The state types are exactly the set of primitive state types and are ranged over by $S, S_1, S_2, \ldots$. Value types are generated by the following grammar:

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

The unit and product types are standard. The value function type $(A_1; S_1) \to (A_2; S_2)$, where S_1 and S_2 are state types, will be interpreted using the closure.

Assume a pair of countably infinite sets of value variables $x, x_1, x_2, \ldots$ and state variables $z, z_1, z_2, \ldots$. Value contexts Γ are defined as a list of variable/type pairs where no variable may appear more than once. State contexts Δ are single variable/type pairs of the form $z : S$.

State Calculus:

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

Value Calculus:

$$\frac{x : A \in \Gamma}{\Gamma \vdash^{\mathsf{v}} x : A}\ (\text{V-VAR}) \qquad \frac{\Gamma \vdash^{\mathsf{v}} e : A_1 \qquad (\mathtt{f} : A_1 \longrightarrow A_2) \in \Phi_V}{\Gamma \vdash^{\mathsf{v}} \mathtt{f}e : A_2}\ (\text{V-PRIM}) \qquad \frac{}{\Gamma \vdash^{\mathsf{v}} \star_1 : 1}\ (\text{V-1I})$$

$$\frac{\Gamma \vdash^{\mathsf{v}} e_1 : A_1 \qquad \Gamma \vdash^{\mathsf{v}} e_2 : A_2}{\Gamma \vdash^{\mathsf{v}} (e_1, e_2) : A_1 \times A_2}\ (\text{V-}\times\text{I}) \qquad \frac{\Gamma \vdash^{\mathsf{v}} e : A_1 \times A_2}{\Gamma \vdash^{\mathsf{v}} \pi_i e : A_i}\ (\text{V-}\times\text{E-}i)$$

$$\frac{\Gamma, x : A_1; z : S_1 \vdash^{\mathsf{c}} c : A_2; S_2}{\Gamma \vdash^{\mathsf{v}} \lambda^{\rightarrow}(x^{A_1}; z^{S_1}).c : (A_1; S_1) \rightarrow (A_2; S_2)}\ (\text{V-}{\rightarrow}\text{I})$$

Command Calculus:

$$\frac{\Gamma \vdash^{\mathsf{v}} e : A \qquad \Delta \vdash^{\mathsf{s}} s : S}{\Gamma; \Delta \vdash^{\mathsf{c}} (e; s) : A; S}\ (\text{C-V-S}) \qquad \frac{\Gamma; \Delta \vdash^{\mathsf{c}} c : A_1; S_1 \qquad (\mathtt{p} : (A_1; S_1) \longrightarrow (A_2; S_2)) \in \Phi_C}{\Gamma; \Delta \vdash^{\mathsf{c}} \mathtt{p}c : A_2; S_2}\ (\text{C-PRIM})$$

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

$$\frac{\Gamma \vdash^{\mathsf{v}} e : (A_1; S_1) \rightarrow (A_2; S_2) \qquad \Gamma; \Delta \vdash^{\mathsf{c}} c : A_1; S_1}{\Gamma; \Delta \vdash^{\mathsf{c}} e@_{\rightarrow}c : A_2; S_2}\ (\text{C-}{\rightarrow}\text{E})$$

FIGURE 1: Typing rules for the Typed Command Calculus

The state calculus is a monadic calculus, in the sense that it only has single-argument functions. The rule S-VAR types variables and S-PRIM types primitive function application. We assume a set Φ_S of primitive state manipulations of the form $\mathtt{m} : S_1 \longrightarrow S_2$.

The value and command calculi are mutually defined via the rules for function abstraction and application. The value calculus includes the standard rules for product introduction and elimination. We also assume a set Φ_V of primitive value functions of the form $\mathtt{f} : A_1 \longrightarrow A_2$. The rule V-$\rightarrow$I introduces functions. Since a function represents a suspended computation, they are treated as pure values and so this rule takes a judgement in the command calculus and produces one in the value calculus. The rule C-$\rightarrow$E eliminates such 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 taken from a set Φ_C of primitive commands of the form $\mathtt{p} : (A; S_1) \longrightarrow (B; S_2)$. The rule C-LET sequences two computations similar to the $M\ \mathtt{to}\ x.N$ construct of the fine-grain call-by-value calculus.

3.2. Example

We present a short example of composable continuations expressed in our calculus, adapted from Wadler's paper [18]. 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 like so[2]:

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

[2]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.

Value calculus:

$$\begin{array}{rcl} \pi_i(e_1, e_2) & = & e_i \\ e & = & (\pi_1 e, \pi_2 e) \\ e & = & \star \\ f & = & (\lambda^{\rightarrow}(x^{A_1}, z^{S_1}).f @_{\rightarrow}(x; z)) \end{array}$$

Command calculus rules:

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

$$\begin{array}{rcl} \text{let } (x; z) \Leftarrow (e; s) \text{ in } c & = & c[e/x, s/z] \\ \text{let } (x; z) \Leftarrow c \text{ in } (x; z) & = & c \\ (\lambda^{\rightarrow}(x, z).c) @_{\rightarrow} a & = & \text{let } (x; z) \Leftarrow a \text{ in } c \\ C[\text{let } (x; z) \Leftarrow c_1 \text{ in } c_2] & = & \text{let } (x; z) \Leftarrow c_1 \text{ in } C[c_2] \end{array}$$

$$C[-] ::= - \mid \mathrm{p}\, C[-] \mid \text{let } (x; z) \Leftarrow C[-] \text{ in } c \mid e @_{\rightarrow} C[-]$$

FIGURE 2: Equational Rules for the Typed Command Calculus

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

let $(x; z) \Leftarrow$ reset
  (let $(y; z) \Leftarrow$ shift f.(let $(a; z) \Leftarrow f @_{\rightarrow}(100; z)$ in let $(b; z) \Leftarrow f @_{\rightarrow}(1000; z)$ in $(a + b; z)$)
   in $(y + 10; z)$)
in $(1 + x; z)$

Given the interpretation of composable continuations above, this term evaluates to 1121. The context let $(y'; z) \Leftarrow -$ in $(y' + 10; z)$ is invoked twice by the application of the delimited continuation exposed by the shift operator. Note that, since the composable continuations example uses a discrete category as the parameterising category, the terms of the state calculus are extremely uninteresting and the variable z representing the continuation context type information is just robotically passed throughout the program. For this application it would make sense to suppress the state calculus, but for more complex examples, such as separated state, it is essential.

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 $(\cdot; \cdot)$ term construct. It also has $\beta\eta$ rules for sequencing, a β rule for functions and a collection of commuting conversions for the sequencing construct.

The rules generate three equational judgements of the form $\Gamma \vdash^{\mathsf{v}} e_1 = e_2 : A$, $\Delta \vdash^{\mathsf{s}} s_1 = s_2 : S$ and $\Gamma; \Delta \vdash^{\mathsf{c}} c_1 = c_2 : A; S$. Note that the equations only apply when both sides are well-typed with the same context and result type. By proving the appropriate substitution rules admissible [1], it is easy to see that all the rules apart from the commuting conversions generate well-typed equations. The commuting conversions can be seen to generate well-typed equations by induction on the structure of the contexts $C[-]$.

3.4. Soundness and Completeness

The interpretation of the Typed Command Calculus in a parameterised Freyd category has already been alluded to, but we spell it out in a bit more detail here. 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 state calculus is interpreted in $\mathcal{S}$, using the identity for the S-VAR 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 $J(\langle id, id\rangle, id); \Gamma \otimes c_1; c_2$. 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 judgements. Completeness is proven by the construction of a closed parameterised Freyd category from the three calculi and the construction of a model within it. See [1] for the more general case of the monoidal Typed Command Calculus (Section 5 below). □

4. MONOIDAL PARAMETERISATION

As we stated in the introduction, the simple parameterisation case is only suitable for modelling situations where the state is global, but in the case of side-effects and typed I/O, we can regard the state as being composed of multiple independent parts. We use symmetric monoidal structure on the state category to represent the composition of state types from smaller state types. The main problem now is to represent the lifting of computations up to larger state contexts in order to compose them. We present two solutions to this problem, one for parameterised monads and one for parameterised Freyd categories, and prove that they are equivalent. For this section, we assume that the parameterising category $\mathcal{S}$ is symmetric monoidal.

4.1. Parameterised Monads with Monoidal Lifting

As mentioned in the introduction, we require two families of arrows $(- \otimes S)^\dagger$ and $(S \otimes -)^\dagger$ to lift computations up to larger state contexts. If the object $T(S_1, S_2, A)$ is interpreted as commands that go from state S_1 to state S_2 and returns a value of type A, then the effect of the monoidal lifting is to lift such commands to larger state contexts: e.g. applying $(- \otimes S)^\dagger$ gives $T(S_1 \otimes S, S_2 \otimes S, A)$. In this respect, the monoidal lifting performs the same service as the strength in lifting computations to larger contexts.

Before we can define the structure we require, we need a way of describing actions on parameterised monads that agree on objects with functors on the parameterising category.

Definition 6 *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 and multiplication of the monad:*

$$F^\dagger; T(FS_1, FS_2, F^\dagger); \mu = \mu; F^\dagger \qquad\qquad \eta; F^\dagger = \eta_F$$

A natural transformation $\zeta : F \Rightarrow G$ in $\mathcal{S}$ is natural for liftings *$F^\dagger$ and $G^\dagger$ if the equation $F^\dagger; T(FS_1, \zeta, A) = G^\dagger; T(\zeta, GS_2, A)$ holds.*

Extending the alternative 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.

Using the given definition, we can state the structure we require on parameterised monads to interpret computation in state context.

Definition 7 *An $\mathcal{S}$-parameterised monad (T, η, μ) has monoidal lifting if, for every $S \in \mathrm{Ob}\mathcal{S}$, there are liftings for the functors $- \otimes S$ and $S \otimes -$, written $(- \otimes S)^\dagger$ and $(S \otimes -)^\dagger$, such that all the symmetric monoidal structure transformations are natural for them and so are the natural transformations given by $- \otimes s$ and $s \otimes -$ for every arrow s.*

4.2. Examples

Two of the examples given above extend to the monoidal case.

4.2.1. Typed Side-effects

For the state category, we take the free symmetric monoidal category on the category $\mathcal{S}$ defined for the typed side-effects example above, which we label $\mathcal{S}'$. We extend the functor $\widehat{\cdot}$ to be $\mathcal{S}' \to \mathcal{C}$ by mapping objects $S_1 \otimes S_2$ to $\widehat{S_1} \times \widehat{S_2}$, hence $\widehat{\cdot}$ becomes a strict symmetric monoidal functor. Monoidal lifting is defined as: $(S \otimes -)^\dagger = c \mapsto \lambda(s, s_1)$.let $(s_2, a) = c(s_1)$ in $((s, s_2), a)$ and similar for $(- \otimes S)^\dagger$.

4.2.2. Typed I/O

Again, we restrict to $\mathcal{C} = \mathrm{Set}$, the category of sets and functions. Choose a small discrete category $\mathcal{S}_o$ to represent the states of individual I/O devices and assume a collection of input sets $\{I(S)\}_{S \in \mathcal{S}_0}$ and a collection of output sets $\{O(S_1, S_2)\}_{S_1, S_2 \in \mathcal{S}_0}$. Set the parameterising category $\mathcal{S}$ to be the free symmetric monoidal category over $\mathcal{S}_0$. That is, objects of $\mathcal{S}$ are words of objects of $\mathcal{S}_0$ formed by unit, singleton and concatenation and arrows are permutations. This category is strict with respect to associativity and units. We use the notation $S(S_1)$ to represent a word (object of $\mathcal{S}$) with a distinguished location in it holding an $\mathcal{S}_0$ object S_1.

The object part of the functor of the monad is built using the following inductive definition:

$$\frac{a \in A}{\mathsf{e}(a) \in T(S, S, A)} \qquad \frac{f \in I(S_1) \Rightarrow T(S(S_1), S_2, A)}{\mathsf{i}(f) \in T(S(S_1), S_2, A)} \qquad \frac{m \in O(S_1, S_2) \qquad c \in T(S(S_2), S, A)}{\mathsf{o}(m, c) \in T(S(S_1), S, A)}$$

$$\frac{c \in T(S_1, S_2, A) \qquad \sigma \text{ a permutation on } S_1}{\mathsf{s}(\sigma, c) \in T(\sigma S_1, S_2, A)}$$

This construction is subject to the following equations:

$$\mathsf{s}(\sigma, \mathsf{i}(f)) = \mathsf{i}(\lambda x.\mathsf{s}(\sigma, fx)) \qquad \mathsf{s}(\sigma, \mathsf{o}(m, c)) = \mathsf{o}(m, \mathsf{s}(\sigma, c)) \qquad \mathsf{s}(\sigma_1, \mathsf{s}(\sigma_2, c)) = \mathsf{s}(\sigma_1; \sigma_2, c)$$

By these equations, it is easy to see that every member of $T(S_1, S_2, A)$ is equal to $\mathsf{s}(\sigma, c)$ for some c not containing a sub-term of the form $\mathsf{s}(\sigma', c')$. Therefore, computations are trees of output transitions-in-context and branching input transitions-in-context, with values at the leaves. The action of the functor on permutations is given by either applying a permutation to the root of the tree, or to all leaves. Monad unit and multiplication are defined as above. Monoidal lifting is defined by appending additional context to the left or right of each node of the tree.

4.3. Double Parameterised Freyd Categories

A double parameterised Freyd category is just a parameterised Freyd category with premonoidal structure with respect to both of the domain categories. This grants us the capability of lifting a computation to a larger state context as well as a larger value context.

State Calculus: rules S-VAR and S-PRIM as for the Typed Command Calculus:

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

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

Value Calculus: as for Typed Command Calculus.

Command Calculus: rules C-V-S, C-PRIM, C-→E as for the Typed Command Calculus.

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

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

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

FIGURE 3: Typing rules for the Monoidal Typed Command Calculus

Definition 8 *A* double parameterised Freyd category *consists of three categories $\mathcal{C}$, $\mathcal{S}$ and $\mathcal{K}$, where $\mathcal{C}$ and $\mathcal{S}$ are symmetric monoidal, and five functors: $J : \mathcal{C} \times \mathcal{S} \to \mathcal{K}$, $\otimes_{\mathcal{C}} : \mathcal{C} \times \mathcal{K} \to \mathcal{K}$, $\otimes_{\mathcal{C}} : \mathcal{K} \times \mathcal{C} \to \mathcal{K}$, $\otimes_{\mathcal{S}} : \mathcal{S} \times \mathcal{K} \to \mathcal{K}$ and $\otimes_{\mathcal{S}} : \mathcal{K} \times \mathcal{S} \to \mathcal{K}$, such that J is identity-on-objects and $(J, \otimes_{\mathcal{C}}, \otimes_{\mathcal{C}})$ and $(J, \otimes_{\mathcal{S}}, \otimes_{\mathcal{S}})$ obey the obvious adaptations of the conditions of Definition 3.*

This 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.

Theorem 3 *For an $\mathcal{S}$-parameterised monad (T, η, μ), given a monoidal lifting we can get a premonoidal structure with respect to $\mathcal{S}$ on $\mathcal{C}_T$ and* vice versa*. These operations are inverse.*

Proof Given a monoidal lifting $(S \otimes -)^\dagger$, define $s \otimes_{\mathcal{S}} c$ as $c; (S_1 \otimes -)^\dagger; T(S_1 \otimes S_1', s \otimes S_2', B)$; the symmetric case is similar. Given premonoidal structure $\otimes_{\mathcal{S}}$, define $(S \otimes -)^\dagger = S \otimes_{\mathcal{S}} id_{T(S_1, S_2, B)}$, the symmetric case is again similar. These obey the required axioms by routine checking. In particular, the two sets of naturality constraints are in one-to-one correspondence. 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 $\mathcal{C}_T$. □

5. MONOIDAL TYPED COMMAND CALCULUS

We now 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 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 appear more than once. We define the joining relation $- \bowtie - = -$ on triples of contexts by the following clauses:

$$I \bowtie I = I \qquad (\Delta_1, x : A) \bowtie \Delta_2 = (\Delta_1 \bowtie \Delta_2), x : A \qquad \Delta_1 \bowtie (\Delta_2, x : A) = (\Delta_1 \bowtie \Delta_2), x : A$$

Given contexts Δ_1, Δ_2, we write $\Delta_1 \bowtie \Delta_2$ to stand for any context Δ_3 such that $\Delta_1 \bowtie \Delta_2 = \Delta_3$.

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

The operations read and store and the state manipulation function clear from the typed side-effects example above can be typed in the calculus as primitive operations like so:

$$\mathtt{read} : (1, \mathtt{[}A\mathtt{]}) \to (A, \mathtt{[}A\mathtt{]}) \qquad \mathtt{store} : (A, \Diamond) \to (1, \mathtt{[}A\mathtt{]}) \qquad \mathtt{clear} : \mathtt{[}A\mathtt{]} \to \Diamond$$

Using these, we can write the following function which takes three storage cells, two containing integers and the third of indeterminate type and adds the contents of the first two, placing the result in the third and forgetting the types of the contents of the first two cells.

$$\begin{array}{l} \lambda(x; z).\text{let } (\star; i, o) \Leftarrow (\star; z) \text{ in let } (\star; i_1, i_2) \Leftarrow (\star; i) \text{ in} \\ \qquad \text{let } (a; i_1) = \mathtt{read}\ (\star; i_1) \text{ in let } (b; i_2) = \text{read } (\star; i_2) \text{ in} \\ \qquad \text{let } (\star; o) = \mathtt{store}\ (a + b; o) \text{ in } (\star; ((\mathtt{clear}\ i_1, \mathtt{clear}\ i_2), o)) \\ : (1, (\mathtt{[Int]} \otimes \mathtt{[Int]}) \otimes \Diamond) \to (1; (\Diamond \otimes \Diamond) \otimes \mathtt{[Int]}) \end{array}$$

5.2. Equational Theory and Soundness and Completeness

The equational rules for the 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 [3], 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.

As before the equational rules generate three equational judgements 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 sketched above,

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: as for the Typed Command Calculus.

Command Calculus:

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

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

$$C[-] \quad = \quad - \mid \mathrm{p}\, C[-] \mid \text{let } (x; z) \Leftarrow C[-] \text{ in } c \mid \text{let } (x; z_1, z_2) \Leftarrow C[-] \text{ in } c \mid \text{let } (x; \star_I) \Leftarrow C[-] \text{ in } c \mid e@_{\rightarrow}C[-]$$

FIGURE 4: Equational Rules for the Monoidal Typed Command Calculus

the equational theory so generated is sound and complete for closed double parameterised Freyd categories. See [1] for the proof.

Theorem 4 *The Monoidal Typed Command Calculus is sound and complete for closed double parameterised Freyd categories.*

6. RELATED WORK

Computational monads [8, 9] have been extremely successful in providing a framework for modelling a large range of computational phenomena, such as dynamic name generation [16]. They have also be used to do effectful programming in pure functional languages [4]. Power and Robinson introduced Freyd Categories [12] as an alternative presentation of strong monads.

Effect Systems [6] augment traditional type systems with information about the side-effects caused by a program's execution. Wadler [19] has presented a connection between effect systems and monads indexed by effect types. The difference between the indexed monads presented by Wadler and our parameterised monads is that the indexed monads are indexed by one variable representing the effects encapsulated by that monad, whereas our parameterisation represents the start and finish states of the effectful computation.

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 typed I/O. By also considering monoidal parameterisation, our definitions also cover separated side-effects and multiple streams of I/O. We have also presented two typed λ-calculi which are sound and complete for the simple parameterisation and monoidal parameterisation cases.

For future work, we intend to consider the effect of indexed types so that the objects of the parameterising category can be refined with “names” of the regions of memory being referred to. This should enhance the categorical structures to be able to interpret type systems such as the Linear Language with Locations [10] and Alias Types [15]. The connection with Hoare logic needs to be further investigated, and there is an obvious analogy between the computation in state context we have considered here and the frame rule of Separation Logic [14].

Finally, Plotkin and Power’s approach of deriving computational monads from algebras of operations and equations [11] 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 [1]).

Acknowledgement This work was funded by the ReQueST grant (EP/C537068) from the Engineering and Physical Sciences Research Council.

REFERENCES

[1] Robert Atkey. *Substructural Simple Type Theories for Separation and In-place Update*. PhD thesis, University of Edinburgh, 2006.

[2] Olivier Danvy and Andrzej Filinski. A functional abstract of typed contexts. Technical report, DIKU – Computer Science Department, University of Copenhagen, August 1989.

[3] Neil Ghani. *Adjoint Rewriting*. PhD thesis, University of Edinburgh, 1995.

[4] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL*, pages 71–84, 1993.

[5] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185:182–210, 2003.

[6] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proceedings of 15th ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[7] Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 2nd edition, 1998.

[8] E. Moggi. Computational lambda-calculus and monads. In Rohit Parikh, editor, *Proceedings of the Fourth Annual IEEE Symp. on Logic in Computer Science, LICS 1989*, pages 14–23. IEEE Computer Society Press, June 1989.

[9] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[10] Greg Morrisett, Amal J. Ahmed, and Matthew Fluet. L^3: A linear language with locations. In *TLCA*, pages 293–307, 2005.

[11] Gordon D. Plotkin and John Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures : 5th International Conference, FOSSACS 2002*, number 2303 in Lecture Notes in Computer Science. Springer-Verlag, April 2002.

[12] John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Math. Struct. in Comp. Science*, 7:453–468, 1997.

[13] John Power and Hayo Thielecke. Closed freyd- and kappa-categories. In *ICALP*, volume 1644 of *Lecture Notes in Computer Science*. Springer, 1999.

[14] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[15] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *ESOP*, pages 366–381, 2000.

[16] Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994.

[17] Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997.

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

[19] Philip Wadler. The marriage of effects and monads. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP ’98)*, pages 63–74, 1998.