Environments, Continuation Semantics and Indexed Categories

John Power* and Hayo Thielecke

Department of Computer Science,
University of Edinburgh,
King's Buildings,
Edinburgh EH9 3JZ,
Scotland, UK
Fax +44 131 667 7209
email: ajp@dcs.ed.ac.uk, ht@dcs.ed.ac.uk

Abstract. There have traditionally been two approaches to modelling environments, one by use of finite products in Cartesian closed categories, the other by use of the base categories of indexed categories with structure. Recently, there have been more general definitions along both of these lines: the first generalising from Cartesian to symmetric premonoidal categories, the second generalising from indexed categories with specified structure to κ-categories. The added generality is not of the purely mathematical kind; in fact it is necessary to extend semantics from the logical calculi studied in, say, Type Theory to more realistic programming language fragments. In this paper, we establish an equivalence between these two recent notions. We then use that equivalence to study semantics for continuations. We give three category theoretic semantics for modelling continuations and show the relationships between them. The first is given by a continuations monad. The second is based on a symmetric premonoidal category with a self-adjoint structure. The third is based on a κ-category with indexed self-adjoint structure. We extend our result about environments to show that the second and third semantics are essentially equivalent, and that they include the first.

1 Introduction

Traditionally in denotational semantics, there have been two categorical ways of modelling contexts and environments. The first is given by finite products in a Cartesian closed category, as for instance in modelling the simply typed λ-calculus. Over the years, that has gradually been extended. For instance, in order to model partiality, one must generalise from finite product structure to symmetric monoidal structure; and more recently, that has been further generalised to the notion of symmetric premonoidal structure [16].

* This work is supported by EPSRC grant GR/J84205: Frameworks for programming language semantics and logic.

A premonoidal category is essentially a monoidal category except that the tensor need only be a functor in two variables separately, and not necessarily a bifunctor: given maps $f : A \to A'$ and $g : B \to B'$, the evident two maps from $A \otimes B$ to $A' \otimes B'$ may differ. Such structures arise naturally in the presence of computational effects, where the difference between these two maps is a result of sensitivity to evaluation order. So that is the structure we need in order to model environments in the presence of continuations or other such strong computational effects (for some examples for this, see Subsection 4.2 below). A program phrase in environment Γ is modelled by a morphism in the premonoidal category with domain $[\![\Gamma]\!]$.

The second approach to modelling environments categorically, also used to model the simply typed λ-calculus, is based on indexed categories with structure, and has been heavily advocated, although not introduced, by Bart Jacobs [8]: the slogan is that contexts, which we call environments, are indices for the categories in which the terms definable in that context are modelled. Here, a program phrase in environment Γ is modelled by an element $\mathbf{1} \longrightarrow [\![\tau]\!]$ in a category that implicitly depends on Γ, i.e., by an arrow from $\mathbf{1}$ to $[\![\tau]\!]$ in the fibre of the indexed category over $[\![\Gamma]\!]$. We consider a weak version of indexed category with structure, called a κ-category, implicit in recent work by Masahito Hasegawa [7]. In the setting of indexed categories, various binding constructs can be studied. A κ-category has a weak first order notion of binding, given by the assertion that reindexing along projections has a left adjoint. In programming terms, that corresponds to a special form that binds an identifier but is not reifying in the sense that it does not produce a *first class* function. Hasegawa [7] compares it to `lambda` in *early* LISP.

The first major result of this paper is to prove the above two models of environments equivalent. More precisely, we show that every symmetric premonoidal category with a little more of the structure cited above, gives rise to a κ-category, and that this gives a bijection between the classes of symmetric premonoidal categories with such structure and κ-categories. The extra structure we need on a symmetric premonoidal category $\mathcal{K}$ is a category with finite products $\mathcal{C}$ and an identity on objects strict symmetric premonoidal functor $J : \mathcal{C} \longrightarrow \mathcal{K}$. At first sight, that may seem a somewhat complex structure, but in fact, as made precise in [15], it is particularly natural category theoretic structure, more so than that of premonoidal structure alone, as it is algebraic structure.

In our semantics for environments, just as in the monads as notions of computations approach, a distinction is made between a category of effect-free morphisms, among them the (denotations of) values, and a category of effectful computations. The former category admits finite products, the latter does not: values can be copied and discarded, while computation cannot (in general). The monads approach makes a very specific design decision of how the category of computations arises from the category with finite products; namely, as the Kleisli category of the monad under consideration. This places a somewhat unfortunate emphasis on how the category of computations is constructed, rather than addressing its structure in its own right.As the category of computations

is what one is actually interested in and about which one has computational intuition, it would be advantageous to put greater emphasis on it. Continuations are special among the computational effects one could consider, as they can be described in terms of structure on the category of computations, that is, in terms of self-adjointness. Moreover, the self-adjointness appears inherently indexed by environments. Hence the possibility to pass back and forth between our two ways of modelling environments may actually facilitate this study of continuation semantics.

Given the two models of environments we have outlined above, we can consider how to model continuations with environments modelled in either of the two ways, and we can consider how the models of continuations compare.

In fact, we go a little beyond that by comparing three recent attempts to incorporate continuations into denotational semantics by means of category theoretic structure. The first has been studied extensively by several people, for instance in Andrzej Filinski's thesis (see [5]). It is based on a monad for continuations: one has a type Ans of answers, and the semantics of a program from τ to σ is given by a function from $[\![\tau]\!]$ to the double exponential $([\![\sigma]\!] \to \mathsf{Ans}) \to \mathsf{Ans}$. The monads approach avoids, to some extent, the question of how to model environments in as much as it reverts to the base category for modelling them. The second, by Hayo Thielecke [18–20], is based on a premonoidal category with self-adjoint structure. A functor $\neg : \mathcal{K}^{\mathrm{op}} \longrightarrow \mathcal{K}$ is called self-adjoint on the left if $\neg^{\mathrm{op}}$ is right adjoint to $\neg$, with the same unit and co-unit. Dually, $\neg$ is called self-adjoint on the right if $\neg^{\mathrm{op}}$ is self-adjoint on the left. Self-adjoint structure corresponds to the idea that for each type τ, there is a continuation type $\neg\tau$ that can accept an input of type τ. The third, which we introduce here, is also being developed by Hayo Thielecke [20]. It is based on a κ-category with added structure, and one again adds a self-adjoint construction. The first of these models is less general than the other two, which are essentially equivalent.

While the first approach to modelling continuations relies on continuations being explicitly given by a double exponentiation monad, the other two approaches avoid any assumptions of *how* continuations are implemented, relying instead on axiomatising a property of continuations that one may take to be fundamental: it axiomatises the existence of a "context switch", more formally, the self adjointness

$$\frac{\neg\sigma \longrightarrow \tau}{\neg\tau \longrightarrow \sigma}$$

of the continuation type constructor $\neg$. These two approaches are distinguished by the way in which one models environments, as explained above.

The second major goal of this paper is to introduce the third of these category theoretic models of continuations and extend our proof of the equivalence between the two ways of modelling environments to show that the second and third models of continuations are essentially equivalent, and that they include the first. We take the equivalence between the second and third models as evidence that modelling continuations by self-adjointness is a robust notion in the sense that it is not overly sensitive to the way one models environments, as one

could model them in two different ways, in each case fitting the self-adjointness into the framework.

The paper is organised as follows. In Section 2, we recall the definitions relating to premonoidal categories, and establish a construction we will need later. In Section 3, we define the notion of κ-category, and give the relationship between κ-categories and symmetric premonoidal categories. Section 4 consists of two parts: first, in Subsection 4.1, we recall a fragment of Standard ML of New Jersey, which we call λ+`callcc`, that we use as our paradigmatic language with continuation primitives; we then argue briefly for the kind of categorical structure we need to model this language in Subsection 4.2. In Section 5, we recall the use of monads for modelling continuations. In Section 6, we define the notion of $\otimes\neg$-category, and show how to model continuations in them. In Section 7, we recall the notion of an indexed $\neg$-category and show how to model continuations with this notion. Finally, in Section 8, we extend the relationship between symmetric premonoidal categories and κ-categories to give the relationship, essentially an equivalence, between $\otimes\neg$-categories and indexed $\neg$-categories.

Related Work

The relationship between symmetric premonoidal categories and κ-categories is related to work by Blute, Cockett, and Seely [1]. Implicit in their work is the construction which, to a symmetric premonoidal category with a little added structure, assigns a κ-category. The latter are closely related to their context categories. Identifying precisely which indexed categories thus arise did not appear in their work.

Filinski [5] pioneered the categorical semantics for continuations. He used a notion of "co-cartesian closure" whereas here we use self-adjointness. This is in line with our second approach to continuation semantics. Assuming finite products to exist in the centre of the semantic category (to be defined) overcomes the difficulty that the subcategory of "total" maps in the sense of [5] is demonstrably *too large* [18,19] to admit products.

Bart Jacobs' thesis [8] championed the view of contexts as "*indices* for the terms and types derivable in that context." We believe this to be relevant not only to type theory but also to the modelling of environments in computer science, and we use it for that purpose in our third approach to continuation semantics.

Ong [11] also uses a fibration to model environments for his categorical formulation of the $\lambda\mu$-calculus [14]. As this calculus is an extension of the call-by-name λ-calculus, Ong can assume every fibre to be Cartesian closed. However, for call-by-value programming languages like ML or Scheme, one cannot assume Cartesian closure. (And even if one were to assume call-by-name, the intended meaning of `callcc` would be less than clear.)

2 Premonoidal categories

In this section, we recall the definitions of premonoidal category and strict premonoidal functor, and symmetries for them, as introduced in [16] and further

studied in [15]. We also develop a basic construction on a premonoidal category that we will need later. A premonoidal category is a generalisation of the concept of monoidal category: it is essentially a monoidal category except that the tensor need only be a functor of two variables and not necessarily be bifunctorial, i.e., given maps $f : A \longrightarrow B$ and $f' : A' \longrightarrow B'$, the evident two maps from $A \otimes A'$ to $B \otimes B'$ may differ.

Historically, for instance for the simply typed λ-calculus, environments have been modelled by finite products. More recently, monoidal structure has sometimes been used, for instance when one wants to incorporate an account of partiality [17]. In the presence of stronger computational effects, an even weaker notion is required. If the computational effects are strong enough for the order of evaluation of $f : A \longrightarrow B$ and $f' : A' \longrightarrow B'$ to be observable, as for instance in the case of continuations [18,19], then the monoidal laws cannot be satisfied. The leading example for us of such stronger computational effects are those given by continuations. However, for a simple example of a premonoidal category that may be used for a crude account of state [16], consider the following.

Example 1. Given a symmetric monoidal category $\mathcal{C}$ together with a specified object S, define the category $\mathcal{K}$ to have the same objects as $\mathcal{C}$, with $\mathcal{K}(A, B) = \mathcal{C}(S \otimes A, S \otimes B)$, and with composition in $\mathcal{K}$ determined by that of $\mathcal{C}$. For any object A of $\mathcal{C}$, one has functors $A \otimes - : \mathcal{K} \longrightarrow \mathcal{K}$ and $- \otimes A : \mathcal{K} \longrightarrow \mathcal{K}$, but they do not satisfy the bifunctoriality condition above, hence do not yield a monoidal structure on $\mathcal{K}$. They do yield a premonoidal structure, as we define below.

In order to make precise the notion of a premonoidal category, we need some auxiliary definitions.

Definition 2. A *binoidal category* is a category $\mathcal{K}$ together with, for each object A of $\mathcal{K}$, functors $h_A : \mathcal{K} \longrightarrow \mathcal{K}$ and $k_A : \mathcal{K} \longrightarrow \mathcal{K}$ such that for each pair (A, B) of objects of $\mathcal{K}$, $h_A B = k_B A$. The joint value is denoted $A \otimes B$.

Definition 3. An arrow $f : A \longrightarrow A'$ in a binoidal category is *central* if for every arrow $g : B \longrightarrow B'$, the following diagrams commute:

$$\begin{array}{ccc} A \otimes B & \xrightarrow{A \otimes g} & A \otimes B' \\ {\scriptstyle f \otimes B}\downarrow & & \downarrow{\scriptstyle f \otimes B'} \\ A' \otimes B & \xrightarrow{A' \otimes g} & A' \otimes B' \end{array} \qquad \begin{array}{ccc} B \otimes A & \xrightarrow{g \otimes A} & B' \otimes A \\ {\scriptstyle B \otimes f}\downarrow & & \downarrow{\scriptstyle B' \otimes f} \\ B \otimes A' & \xrightarrow{g \otimes A'} & B' \otimes A' \end{array}$$

Moreover, given a binoidal category $\mathcal{K}$, a natural transformation $\alpha : g \Longrightarrow h : \mathcal{B} \longrightarrow \mathcal{K}$ is called *central* if every component of α is central.

Definition 4. A *premonoidal category* is a binoidal category $\mathcal{K}$ together with an object I of $\mathcal{K}$, and central natural isomorphisms a with components $(A \otimes B) \otimes C \longrightarrow A \otimes (B \otimes C)$, l with components $A \longrightarrow A \otimes I$, and r with components $A \longrightarrow I \otimes A$, subject to two equations: the pentagon expressing coherence of a, and the triangle expressing coherence of l and r with respect to a.

Now we have the definition of a premonoidal category, it is routine to verify that Example 1 is an example of one. There is a general construction that yields premonoidal categories too: given a strong monad T on a symmetric monoidal category $\mathcal{C}$, the Kleisli category $\mathbf{Kleisli}(T)$ for T is always a premoidal category, with the functor from $\mathcal{C}$ to $\mathbf{Kleisli}(T)$ preserving premonoidal structure strictly: of course, a monoidal category such as $\mathcal{C}$ is trivially a premonoidal category. That construction is fundamental, albeit implicit, in Eugenio Moggi's work on monads as notions of computation [10], as explained in [16].

Definition 5. Given a premonoidal category $\mathcal{K}$, define the *centre* of $\mathcal{K}$, denoted $\mathsf{Z}(\mathcal{K})$, to be the subcategory of $\mathcal{K}$ consisting of all the objects of $\mathcal{K}$ and the central morphisms.

For an example of the centre of a premonoidal category, consider Example 1 for the case of $\mathcal{C}$ being the category Set of small sets, with symmetric monoidal structure given by finite products. Suppose S has at least two elements. Then the centre of $\mathcal{K}$ is precisely Set. In general, given a strong monad on a symmetric monoidal category, the base category $\mathcal{C}$ need not be the centre of $\mathbf{Kleisli}(T)$, but, modulo a faithfulness condition sometimes called the mono requirement [10,16], must be a subcategory of the centre.

The functors h_A and k_A preserve central maps. So we have

Proposition 6. *The centre of a premonoidal category is a monoidal category.*

This proposition allows us to prove a coherence result for premonoidal categories, directly generalising the usual coherence result for monoidal categories. Details appear in [16].

Definition 7. A *symmetry* for a premonoidal category is a central natural isomorphism with components $c : A \otimes B \longrightarrow B \otimes A$, satisfying the two conditions $c^2 = 1$ and equality of the evident two maps from $(A \otimes B) \otimes C$ to $C \otimes (A \otimes B)$. A *symmetric* premonoidal category is a premonoidal category together with a symmetry.

All of the examples of premonoidal categories we have discussed so far are symmetric, and in fact, symmetric premonoidal categories are those of primary interest to us, and seem to be those of primary interest in denotational semantics in general. For an example of a premonoidal category that is not symmetric, consider, given any category $\mathcal{C}$, the category $End_u(\mathcal{C})$ whose objects are functors from $\mathcal{C}$ to itself, and for which an arrow from h to k is a $\mathcal{C}$-indexed family of arrows $\alpha(A) : h(A) \longrightarrow k(A)$ in $\mathcal{C}$, i.e., what would be a natural transformation from h to k but without assuming commutativity of the naturality squares. Then, this category, together with the usual composition of functors, has the structure of a strict premonoidal category, i.e., a premonoidal category in which all the structural isomorphisms are identities, which is certainly not symmetric.

Definition 8. A *strict premonoidal functor* is a functor that preserves all the structure and sends central maps to central maps.

One may similarly generalise the definition of strict symmetric monoidal functor to strict symmetric premonoidal functor.

In order to compare the various models of environments in the next section, we need to study a construction that, to a premonoidal category, assigns a $\mathcal{C}at$-valued functor.

Definition 9. A *comonoid* in a premonoidal category $\mathcal{K}$ consists of an object $\mathbf{C}$ of $\mathcal{K}$, and central maps $\delta : \mathbf{C} \longrightarrow \mathbf{C} \otimes \mathbf{C}$ and $\nu : \mathbf{C} \longrightarrow I$ making the usual associativity and unit diagrams commute.

It follows from centrality of the two maps in the definition of comonoid that one has the usual coherence for a comonoid, i.e., n-fold associativity is well defined, and comultiple products with counits are also well defined.

Definition 10. A *comonoid map* from $\mathbf{C}$ to $\mathbf{D}$ in a premonoidal category $\mathcal{K}$ is a central map $f : \mathbf{C} \longrightarrow \mathbf{D}$ that commutes with the comultiplications and counits of the comonoids.

Again, it follows from centrality that a comonoid map preserves multiple application of comultiplication and counits. Given a premonoidal category $\mathcal{K}$, comonoids and comonoid maps in $\mathcal{K}$ form a category $\mathbf{Comon}(\mathcal{K})$ with composition given by that of $\mathcal{K}$. Moreover, any strict premonoidal functor sends a comonoid to a comonoid, so any strict premonoidal functor $H : \mathcal{K} \longrightarrow \mathcal{L}$ lifts to a functor $\mathbf{Comon}(H) : \mathbf{Comon}(\mathcal{K}) \longrightarrow (\mathcal{L})$.

Trivially, any comonoid $\mathbf{C}$ in a premonoidal category $\mathcal{K}$ yields a comonad on $\mathcal{K}$ given by $- \otimes \mathbf{C}$, and any comonoid map $f : \mathbf{C} \longrightarrow \mathbf{D}$ yields a map of comonads from $- \otimes \mathbf{C}$ to $- \otimes \mathbf{D}$, and hence a functor from $\mathbf{Kleisli}(- \otimes \mathbf{D})$, the Kleisli category of the comonad $- \otimes \mathbf{D}$, to $\mathbf{Kleisli}(- \otimes \mathbf{C})$, that is the identity on objects. So we have a functor from $\mathbf{Comon}(\mathcal{C})^{\mathrm{op}}$ to $\mathcal{C}at$, which we denote by $\mathsf{s}(\mathcal{K})$. See [16] for this construction and another application of it.

Now, given a category $\mathcal{C}$ with finite products, every object A of $\mathcal{C}$ has a unique comonoid structure, given by the diagonal and the unique map to the terminal object. So $\mathbf{Comon}(\mathcal{C})$ is isomorphic to $\mathcal{C}$.

Thus, given a category $\mathcal{C}$ with finite products, a premonoidal category $\mathcal{K}$, and a strict premonoidal functor $J : \mathcal{C} \longrightarrow \mathcal{K}$, we have a functor $\kappa(J) : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ given by $\mathsf{s}(\mathcal{K})$ composed with the functor induced by J from $\mathcal{C} \cong \mathbf{Comon}(\mathcal{C})$ to $\mathbf{Comon}(\mathcal{K})$.

3 κ-categories

In this section, we introduce κ-categories, and show that the construction at the end of Section 2 yields an equivalence between premonoidal categories with added structure as we shall make precise, and κ-categories.

Hasegawa has decomposed the λ-calculus into two calculi, the κ-calculus, and the ζ-calculus [7]. This analysis arose from study of Hagino's categorical programming language. The idea of the κ-calculus, also known as the contextual

calculus, is that it has product types on which its abstraction and reduction are constructed, and it can be regarded as a reformulation of the first-order fragment of simply-typed λ-calculus, but does not require the exponent types. We do not explicitly present the κ-calculus here. However, we do describe the notion of κ-category, which is a categorical analogue of the definition of κ-calculus. Further, we compare the notion of κ-category with that of symmetric premonoidal category with a extra structure. That relationship is one of the main theorems of the paper, which we later extend to relate our two main models of continuations.

Given a small category $\mathcal{C}$, a functor from $\mathcal{C}^{\mathrm{op}}$ to Cat is called an *indexed category*, a natural transformation between two indexed categories is called an *indexed functor*. The notion of *indexed natural transformation* is definable too, and this gives us a evident notion of adjunction between indexed categories. In concrete terms, it amounts to an $\mathbf{Ob}\,\mathcal{C}$-indexed family of adjunctions, such that the units and counits are preserved by reindexing along each $f : A \longrightarrow B$. And given an indexed category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$, we denote by $H^{\mathrm{op}} : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ the indexed functor for which $H^{\mathrm{op}}_A = (H_A)^{\mathrm{op}}$ with H^{op}_f defined by H_f.

We will need the definitions of H^{op} and adjunctions between indexed categories in later sections to extend the notion of a functor being self-adjoint on the left, as in the semantics for continuations with premonoidal structure used to model environments in Section 6 to that of an *indexed* functor being self-adjoint on the left as in the semantics for continutations using κ-categories to model environments in Section 7. But now for our definition of κ-category.

Definition 11. A *κ-category* consists of a small category $\mathcal{C}$ with finite products, together with an indexed category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ such that

- for each object A of $\mathcal{C}$, $\mathbf{Ob}\, H_A = \mathbf{Ob}\,\mathcal{C}$, and for each arrow $f : A \longrightarrow B$ in $\mathcal{C}$, the functor $H_f : H_B \longrightarrow H_A$ is the identity on objects
- for each projection $\pi : B \times A \longrightarrow B$ in $\mathcal{C}$, the functor H_π has a left adjoint L_B given on objects by $- \times A$
- *(the Beck-Chevalley condition)* for every arrow $f : B \longrightarrow B'$ in $\mathcal{C}$, the natural transformation from $L_B \circ H_{f \times id_A}$ to $H_f \circ L_{B'}$ induced by the adjointness is an isomorphism.

$$\begin{array}{ccc} H_{B' \times A} & \xrightarrow{L_{B'}} & H_{B'} \\ {\scriptstyle H_{f \times \mathrm{id}_A}}\downarrow & \Longrightarrow & \downarrow{\scriptstyle H_f} \\ H_{B \times A} & \xrightarrow{L_B} & H_B \end{array}$$

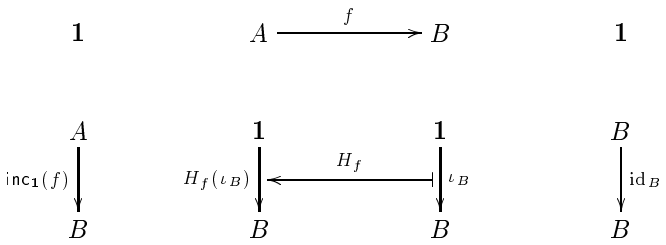
We shall denote the isomorphism associated with the adjunctions given in the definition by

$$\kappa : H_{B \times A}(C, C') \cong H_B(C \times A, C').$$

A κ-category allows us to model the environments in the presence of continuations or other computational effects. Of course, modelling computational

effects involves more structure than that of a κ-category: for continuations, it requires the assignment to each type τ of a type $\neg\tau$ that awaits an input of type τ. We shall study such structure in Section 7, where we shall define an indexed $\neg$-category. But here, we restrict out attention to modelling environments, and we shall pursue our leading example, that of continuations, later.

Proposition 12. *Given a κ-category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$, there is an indexed functor $\mathsf{inc} : \mathsf{s}(\mathcal{C}) \longrightarrow H$ as follows: for each A in $\mathcal{C}$, we have a functor from $\mathsf{s}(\mathcal{C}_A)$ to H_A. On objects, it is the identity. To define inc_1 on arrows, given $f : A \longrightarrow B$ in $\mathcal{C}$, consider the arrow $\iota_B : 1 \longrightarrow B$ in H_B corresponding under the adjunction to id_B in H_1. Applying H_f to it gives a map $H_f(\iota_B) : 1 \longrightarrow B$ in H_A, or equivalently, under the adjunction, a map from A to B in H_1. Define $\mathsf{inc}_1(f)$ to be that map.*



This plus naturality determines the rest of the structure.

Proof. It is immediate that inc_1 preserves identities, and one can prove that it preserves composition: this follows by proving that for any map $f : A \longrightarrow B$ in $\mathcal{C}$ and any map $g : 1 \longrightarrow C$ in H_B, the map $H_f(g)$ corresponds to the composite in H_1 of $\mathsf{inc}_1(f)$ with the adjoint correspondent to g. Moreover, this yields a functor inc_A for every A, with naturality as required. □

Using proposition 12, we can exhibit the relationship between symmetric premonoidal categories with specified extra structure and κ-categories. This forms the basis for the first main result of the paper, Theorem 14. First, for the construction of a κ-category from a symmetric premonoidal category, we have

Proposition 13. *Given a small category $\mathcal{C}$ with finite products, a small symmetric premonoidal category $\mathcal{K}$ and an identity on objects strict symmetric premonoidal functor $J : \mathcal{C} \longrightarrow \mathcal{K}$, the functor $\kappa(J) : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ is a κ-category.*

Proof. It follows immediately from the construction of $\kappa(J)$ in Section 2 that for each object A of $\mathcal{C}$, we have $\mathbf{Ob}\,\kappa(J)_A = \mathbf{Ob}\,\mathcal{C}$, and that for each arrow $f : A \longrightarrow B$ in $\mathcal{C}$, the functor $\kappa(J)_f$ is the identity on objects. Moreover, the existence of the adjoints to each $\kappa(J)_\pi$ follows directly from the construction and the fact that $\mathcal{C}$ is symmetric. The Beck-Chevalley condition also follows directly from the construction. □

Now, for the converse, giving our first main result of the paper.

Theorem 14. *Let $\mathcal{C}$ be a small category with finite products. Given a κ-category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$, there are a symmetric premonoidal category $\mathcal{K}$ and an identity on objects strict symmetric premonoidal functor $J : \mathcal{C} \longrightarrow \mathcal{K}$, unique up to isomorphism, for which H is isomorphic to $\kappa(J)$.*

Proof. Define $\mathcal{K}$ to be H_1. For each object A of $\mathcal{K}$, equally A an object of $\mathcal{C}$ since $\mathbf{Ob}\, H_1 = \mathbf{Ob}\mathcal{C}$, define $- \otimes A : \mathcal{K} \longrightarrow \mathcal{K}$ by the composite $L \circ H_!$ where $! : A \longrightarrow 1$ is the unique map in $\mathcal{C}$ from A to 1. Note that $!$ is of the form π, so the left adjoint exists. Moreover, for each map $g : C \longrightarrow C'$ in $\mathcal{K}$, we have $g \otimes A : C \times A \longrightarrow C' \times A$. The rest of the data and axioms to make $\mathcal{K}$ a symmetric premonoidal category arise by routine calculation, using the symmetric monoidal structure of $\mathcal{C}$ determined by its finite product structure, and by use of the Beck-Chevalley condition.

Define $J : \mathcal{C} \longrightarrow \mathcal{K}$ by inc_1 as in proposition 12. It follows from the Beck-Chevalley condition that for a map $f : A \longrightarrow B$ in $\mathcal{C}$, and for a map $g : C \longrightarrow D$ in H_B, we have that $H_f(g)$ is given by the composite of $J(id_C \times f)$ with the adjoint correspondent of g. The Beck-Chevalley condition further implies that $(\mathsf{inc}_1 -) \otimes A$ agrees with $\mathsf{inc}_1(- \times A)$. It follows from functoriality of the H_f's that every map in $\mathcal{C}$ is sent into the centre of $\mathcal{K}$. Functoriality plus the Beck-Chevalley condition similarly imply that all the structural maps are preserved. So J is an identity on objects strict symmetric premonoidal functor.

It follows directly from our construction of J that $\kappa(J)$ is isomorphic to H. Moreover, $J : \mathcal{C} \longrightarrow \mathcal{K}$ is fully determined by H since $\mathcal{C}$ is fixed, $\mathcal{K}$ must be H_1 up to isomorphism, with premonoidal structure as given, and J must agree on maps with the construction as we have given it. Hence, J is unique up to isomorphism. □

4 Continuations

We recall first-class continuations, and argue informally why their semantics leads us to the categorical notions of premonoidal and self-adjoint structure.

4.1 First-class continuations in ML

We will define a language λ+`callcc` as our paradigmatic language. It is based on an idealised version of **ML** and is given as follows.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}$$

$$\frac{\Gamma \vdash M : \neg\tau \to \tau}{\Gamma \vdash \texttt{callcc}\ M : \tau} \qquad \frac{\Gamma \vdash M : \neg\tau \quad \Gamma \vdash N : \tau}{\Gamma \vdash \texttt{throw}\ M\ N : \sigma}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

Here $\neg$ is the continuation type constructor, its ASCII representation in actual ML programs being `cont`. See [12] for a discussion of this language, including an operational semantics.

As an example of the use of expression continuations in programming, we consider the function `rember-upto-last` from the recent programming textbook *The Seasoned Schemer* [4]:

> The function `rember-upto-last` takes an atom a and a lat [list of atoms] and removes all the atoms from the lat up to and including the last occurrence of a. If there are no occurrences of a, `rember-upto-last` returns the list.

We can compute `rember-upto-last` by recurring over the list and jumping every time the element `a` is encountered, thereby ignoring everything that precedes it in the list. Unlike the *Seasoned Schemer*'s solution, the solutioin presented here does not copy the list while recurring over it:

```
fun remberuptolast a lat =
    callcc(fn skip =>
           let fun R [] = ()
                 | R (b::l) =
                   (R l;
                    if b = a then throw skip l else ())
           in
               (R lat; lat) end);
```

4.2 Motivation of premonoidal categories and self-adjointness

The tuple (similarly, list) notation present in many programming languages may, at first sight, suggest that the appropriate semantic setting ought to be a Cartesian or at least monoidal category.

But in terms of evaluation in a call-by-value language, a tuple (M, N) means that each component has to be evaluated.

This can be made explicit by naming the intermediate values. If the first component is to be evaluated first, one would write:

$$\texttt{let}\ x = M\ \texttt{in}\ \texttt{let}\ y = N\ \texttt{in}\ (x, y)$$

Conversely, to evaluate the second component first, one writes:

$$\texttt{let}\ y = N\ \texttt{in}\ \texttt{let}\ x = M\ \texttt{in}\ (x, y)$$

The `let`-notation, then, has the advantage that the implicit sequencing is made explicit in the textual representation.

For example, in a language with state, there are two possible meanings of a tuple (M, N), depending which component is evaluated first. Consider the following examples, where we make the evaluation order explict by using `let`.

```
let val s = ref 0 in
let val x = (s := !s + 1; !s) in
let val y = (s := !s + 1; !s)
in #1(x,y) end end end ;

let val s = ref 0 in
let val y = (s := !s + 1; !s) in
let val x = (s := !s + 1; !s)
in #1(x,y) end end end;
```

Just as for state, in the presence of continuations (first-class or otherwise) there are two possible meanings of the tuple `(throw k 1, throw k 2)`.

```
callcc(fn k =>
       let val x = throw k 1 in
       let val y = throw k 2
       in #1(x,y) end end);

callcc(fn k =>
       let val y = throw k 2 in
       let val x = throw k 1
       in #1(x,y) end end);
```

In a monoidal category, there would be no way to distinguish between the two composites. This makes monoidal categories suitable for those cases where both composites are evaluated in parallel *or* where there can be no interference between the two (which would the case, say, if both had access to disjoint pieces of state). But with control, as given by continuations, we have both a sequential evaluation order and interference between the components, since a jump in one will prevent the other from being evaluated at all.

Put differently, the presence of computational effects, like state and control, "breaks" the bifunctoriality, so one is left with a binoidal category. (Partiality appears to be a separate case that should perhaps not be lumped together with genuine effects like state and control.)

Next, we illustrate the self-adjointness by some ML code. The reader may find it helpful to look at the types and see what categorical structure the typings correspond to. For the self-adjointness on the left, we define (terms representing) the unit of adjunction, the isomorphism of adjunction and the continuation functor as follows.

```
fun force h = callcc(throw h);
    force : '1a cont cont -> '1a;

fun phi f h = callcc((throw h) o f);
    phi : ('2a cont -> 'b) -> ('b cont -> '2a);

fun negate f = phi(f o force);
    negate : ('1a -> 'b) -> ('b cont -> '1a cont);
```

Complementary (though not dually) to the double-negation map `force`, there is a double negation-introduction map `thunk`.

```
fun thunk a = callcc(fn k => throw (force k) a);
    thunk : 'a -> 'a cont cont;
```

To explain the computational meaning of the self-adjoint structure, we recall the notions of thunking and forcing [13].

In the present setting, we consider a *thunk* to be something that expects a continuation to which it is to pass its argument.

`force` passes the current continuation to its argument; thus, in

$$\texttt{force}(\texttt{thunk}a)$$

the current continuation is passed to `thunk a` so that `a` is passed to the current continuation. Hence `force(thunk a)` = `a`.

Finally, we point out that in the presence of first-class continuations, functions can be identified with (a special case of) continuations by virtue of the following conversions.

```
fun conttofun c a = callcc(fn k => throw c (a,k));
    conttofun : ('a * '2b cont) cont -> ('a -> '2b);

fun funtocont f = callcc((fn (a,k) => throw k (f a)) o force);
    funtocont : ('1a -> '1b) -> ('1a * '1b cont) cont;
```

5 Continuation semantics and monads

The canonical way of giving continuation semantics is by a CPS transform. We recall here a variant of the transform in [3]. [12]

$$\begin{aligned}
[\![x]\!] &= \lambda k.kx \\
[\![\lambda x.M]\!] &= \lambda k.k(\lambda xh.[\![M]\!]h) \\
[\![\texttt{throw}\ M\ N]\!] &= \lambda k.[\![M]\!][\![N]\!] \\
[\![\texttt{callcc}\ M]\!] &= \lambda k.[\![M]\!](\lambda f.fkk) \\
[\![MN]\!] &= \lambda k.\big([\![M]\!](\lambda m.[\![N]\!](\lambda n.mnk))\big)
\end{aligned}$$

The CPS transform can be read as a semantics in the style of monads as notions of computation [10]

The denotation

$$[\![\Gamma \vdash M : \tau]\!] \ldots k$$

of a program phrase of type τ takes an argument k of type $\tau \to \mathsf{Ans}$. Reading this in an uncurried fashion gives rise to a semantics in the monads as notions of computation style

$$[\![\Gamma \vdash \_ : \tau]\!] : [\![\Gamma]\!] \longrightarrow ([\![\tau]\!] \to \mathsf{Ans}) \to \mathsf{Ans}$$

The "monads as notions of computations"-style categorical continuation semantics is then an almost automatic byproduct of the CPS transform: one need only interpret the λ-terms in the "output" of the transform in a Cartesian closed category.

However, this approach runs into conceptual and mathematical difficulties when trying to address the question what the elusive answer type could be [9]. In the light of work on CPS in the π-calculus, such as [2], the answer type seems to be a red herring, in that one can have continuations without an answer type.

The more recent approaches to modelling continuations in category theoretic terms do not have an answer type, so appear to be more general. They merely have, for each type τ, an object $\neg\tau$ that is to be seen as awaiting an input of type τ, but with no commitment to a particular type for the output. Thus correspondingly, the category theoretic structures for semantics are more general too. Much is gained from that added generality in that it can be used to give a more subtle, natural notion of value (see [18,19]).

6 Continuation semantics in $\otimes\neg$-categories

In this section, we use the notion of symmetric premonoidal category together with that of self-adjointness on the left, which we shall define, to give the notion of $\otimes\neg$-category, which we shall use to model continuations. It is routine to verify that, for the monad $(- \to \mathsf{Ans}) \to \mathsf{Ans}$ of Section 5, the Kleisli category $\mathbf{Kleisli}(- \to \mathsf{Ans}) \to \mathsf{Ans}$ has the structure of a $\otimes\neg$-category, and that the semantics we give here agrees with that there. Even if every $\otimes\neg$- category would be so obtained, that would be a representation theorem which would not detract from the axiomatic view.

First, for the definition of self-adjointness on the left,

Definition 15. A functor $g : \mathcal{K}^{op} \longrightarrow \mathcal{K}$ is called *self-adjoint on the left* if g^{op} is right adjoint to g with the unit of the adjunction and the counit being the same.

To proceed, we need a piece of notation. In a premonoidal category with a subcategory containing the same objects, where the premonoidal structure agrees with finite product structure, we extend the notation for the pairing map to the whole category by convention as follows: given maps $f : A \longrightarrow B$ and $g : A \longrightarrow C$, we write $\langle f, g\rangle : A \longrightarrow B \otimes C$ for

$$A \xrightarrow{\langle \mathsf{id},\mathsf{id}\rangle} A \otimes A \xrightarrow{A\otimes g} A \otimes C \xrightarrow{f\otimes C} B \otimes C$$

Here we have made the (essentially arbitrary) choice that the second component of a tuple is to be evaluated first.

Definition 16. A $\otimes\neg$-*category* consists of a symmetric premonoidal category $\mathcal{K}$ for which, when restricted to $\mathsf{Z}(\mathcal{K})$ with inclusion inc, the premonoidal structure is given by finite products, together with

– a functor $\neg : \mathcal{K}^{\mathrm{op}} \longrightarrow \mathsf{Z}(\mathcal{K})$ such that for each object A of $\mathcal{K}$, $\mathsf{inc} \circ (A \otimes \neg(\_)) : \mathcal{K}^{\mathrm{op}} \longrightarrow \mathcal{K}$ is self-adjoint on the left, and
– a coretract $\mathsf{thunk} : \mathsf{id}_{\mathsf{Z}(\mathcal{K})} \longrightarrow \neg\neg$ in $\mathsf{Z}(\mathcal{K})$ of $\mathsf{apply}_{\mathbf{1}} \circ \mathsf{inc} : \neg\neg\mathsf{inc} \longrightarrow \mathsf{id}_{\mathsf{Z}(\mathcal{K})}$, where apply is the unit of the self-adjunction,

such that

– apply is dinatural in A and
– putting $\underline{\underline{\mathsf{apply}}}_A \stackrel{\mathrm{def}}{=} A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}); \mathsf{apply}_A$, we have

$$\begin{aligned}
\neg\mathsf{apply}_{\mathbf{1}} &= \mathsf{thunk}_{\neg} \\
\mathsf{thunk}; \neg\neg\underline{\underline{\mathsf{apply}}} &= \underline{\underline{\mathsf{apply}}}; \mathsf{thunk} \\
\mathsf{thunk}_{A\otimes C} &= A \otimes \mathsf{thunk}_C; A \otimes \neg\underline{\underline{\mathsf{apply}}}; \underline{\underline{\mathsf{apply}}} \\
\mathsf{apply}_{A\otimes A'} &= \langle \pi_2, \pi_1 \rangle \otimes \neg(A \otimes A' \otimes \neg B); A' \otimes \underline{\underline{\mathsf{apply}}}_A ; \mathsf{apply}_{A'}
\end{aligned}$$

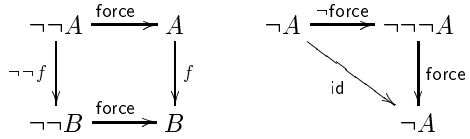
The thinking behind the definition of $\otimes\neg$-category is as follows. First, one needs the premonoidal structure to model first order constructs such as environments and tuple types. On top of this, the continuation type constructor is added as a contravariant functor. The reason for the contravariance is that a program $\sigma \to \tau$ gives rise to a "continuation transformer" $\neg\tau \to \neg\sigma$ by precomposition. Intuitively, this is similar to building a function closure, and that is why $\neg f$ lies in the centre of the premonoidal category. We take central maps as effect-free maps. (See [18,19] for a detailed analysis of this and its advantages.) As explained in the introduction, the self-adjointness on the left accounts for context switch. Its unit corresponds to forcing a thunk [13]. However, this by itself is not sufficient as we also need to pass arguments along with a jump. That is why we require the self-adjointness even in the parameterized sense, i.e., for the functor $A \otimes \neg -$. The unit of this self-adjointness is the call-by-value application map, or jump with arguments. The self-adjointness on the left only ever allows one to eliminate double negations; in order to construct λ-abstractions, one also needs a double negation introduction, or generic closure building operation thunk satisfying $\mathsf{thunk}; \mathsf{apply}_{\mathbf{1}} = \mathsf{id}$.

Dinaturality of apply seems to be fundamental, and is used extensively. The first of the remaining axioms is a fact about continuations. The second is used to make $\underline{\underline{\mathsf{apply}}}$ behave like a value in the sense that if thunk is natural with respect to some map, then that map is somehow value-like. It is like a form of an η law without arguments. The remaining two axioms allow us to relate apply and thunk for different indices.

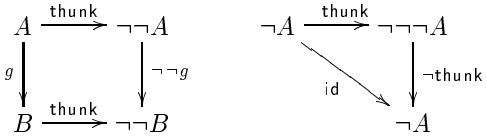
Intuitively, dinaturality of the application map means that modifying the operand of a function application by a map $f : A \longrightarrow A'$ is the same as modifying the operator by a corresponding continuation transformer.

$$\begin{array}{ccc}
A \otimes \neg(A' \otimes \neg B) & \xrightarrow{A\otimes\neg(f\otimes\neg B)} & A \otimes \neg(A \otimes \neg B) \\
{\scriptstyle f\otimes\neg(A'\otimes\neg B)}\downarrow & & \downarrow{\scriptstyle \mathsf{apply}} \\
A' \otimes \neg(A' \otimes \neg B) & \xrightarrow{\mathsf{apply}} & B
\end{array}$$

The universal property of the continuation functor can be expressed by the following diagrams (naturality and triangular identity for force.)



In addition to the usual thunk; force = id, we have another axiom linking forcing and thunking. A consequence of this is the self-adjointness on the right of the restriction of ¬ to the centre, with unit thunk.



(where g is central.)

In a $\otimes\neg$-category, we have a (call-by-value) λ-abstraction by defining

$$\overline{\lambda}_A \stackrel{\text{def}}{=} \mathsf{thunk}; \neg\underline{\underline{\mathsf{apply}}}_A ; A \otimes \neg f$$

Given a $\otimes\neg$-category $\mathcal{K}$, we can give an interpretation $[\![-]\!]$ for λ+`callcc` as follows. Types and environments are interpreted as usual, except for the breaking down of arrow types.

$$\begin{aligned}
[\![\neg\tau]\!] &\stackrel{\text{def}}{=} \neg[\![\tau]\!] \\
[\![\sigma \to \tau]\!] &\stackrel{\text{def}}{=} \neg([\![\sigma]\!] \otimes \neg[\![\tau]\!]) \\
[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] &\stackrel{\text{def}}{=} [\![\tau_1]\!] \otimes \cdots \otimes [\![\tau_n]\!]
\end{aligned}$$

A judgement $\Gamma \vdash M : \tau$ denotes a morphism $[\![\Gamma]\!] \longrightarrow [\![\tau]\!]$, defined by induction on M.

$$\begin{aligned}
[\![x_1 : \tau_1, \ldots x_n : \tau_n \vdash x_j : \tau_j]\!] &\stackrel{\text{def}}{=} \pi_j \\
[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] &\stackrel{\text{def}}{=} \overline{\lambda}_{[\![\sigma]\!]}\, [\![x : \sigma, \Gamma \vdash M : \tau]\!] \\
[\![\Gamma \vdash \mathtt{throw}\ M\ N : \sigma]\!] &\stackrel{\text{def}}{=} \langle [\![\Gamma \vdash M : \neg\tau]\!], [\![\Gamma \vdash N : \tau]\!] \rangle; [\![\tau]\!] \otimes \neg\pi_1; \mathsf{apply} \\
[\![\Gamma \vdash \mathtt{callcc}\ M : \tau]\!] &\stackrel{\text{def}}{=} [\![\Gamma \vdash M : \neg\tau \to \tau]\!]; \neg\langle \mathsf{id}_{\neg[\![\tau]\!]}, \mathsf{id}_{\neg[\![\tau]\!]} \rangle; (\mathsf{apply}_{\mathbf{1}})_{[\![\tau]\!]} \\
[\![\Gamma \vdash MN : \tau]\!] &\stackrel{\text{def}}{=} \langle [\![\Gamma \vdash M : \sigma \to \tau]\!], [\![\Gamma \vdash N : \sigma]\!] \rangle; \mathsf{apply}
\end{aligned}$$

This semantics validates the the *call-by-value* β law $(\lambda x.M)\, V = M[x \mapsto V]$ where V is a value, i.e. a variable or an abstraction. It is worth noting, too, what does not hold. We have neither the unrestricted β law nor equivalences like

$$(\lambda x.\lambda y.L)\, M\, N = (\lambda y.\lambda x.L)\, N\, M$$

because our semantic category is neither closed nor monoidal. Nonetheless, the call-by-value β law could be stated as an adjunction: instaed of closure, we have the weaker notion of central closure [15,18].

A discussion and validation of this semantics as such is beyond the scope of the present paper. To some extent this is ongoing research, while some of it has appeared in [18,19].

In this paper, what matters about the semantics is that it is based upon of the continuation functor *fitted into a framework for the semantics of environments.* This aspect is what we attempt to analyse by varying this framework — at least in terms of presentation.

7 Continuation semantics in indexed $\neg$-categories

In this section, we use the definition of κ-category as a basis, together with self-adjointness, for defining the notion of an indexed $\neg$-category. We then use that latter definition to give our third continuations semantics. In the final section, we shall prove that it is essentially equivalent to the second, i.e., that given by $\otimes\neg$-categories.

Definition 17. An *indexed $\neg$-category* consists of a κ-category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ together with an indexed functor $\neg : H^{\mathrm{op}} \longrightarrow \mathsf{s}(\mathcal{C})$ such that $\mathsf{inc} \circ \neg$ is self-adjoint on the left, together with a coretract thunk of $\mathsf{force}_{\mathbf{1}} \circ \mathsf{inc}$, where force is the unit of the self-adjunction, such that

- force is dinatural in A with respect to all maps in $H_{\mathbf{1}}$ and
- letting $(\underline{\underline{\mathsf{force}}}_A)_B$ be the correspondent under the adjunction to $L_A H_\pi((\mathsf{force}_{\mathbf{1}})_B)$, we have

$$\begin{aligned}
\neg\mathsf{force}_{\mathbf{1}} &= \mathsf{thunk}_\neg \\
\mathsf{thunk}; \neg\neg\underline{\underline{\mathsf{force}}} &= \underline{\underline{\mathsf{force}}}; \mathsf{thunk} \\
\mathsf{thunk}_{A\times C} &= A \times \mathsf{thunk}_C; A \times \neg\underline{\underline{\mathsf{force}}}; \underline{\underline{\mathsf{force}}} \\
\neg\kappa^{-1}(\mathsf{id}_{C\times A}) &= \neg_C(LH_!(\mathsf{force}_{\mathbf{1}})); \mathsf{force}_C
\end{aligned}$$

The left adjoint to reindexing along projections gives rise to a comonad on each fibre, which we will write as $(\_) \otimes A$. Furthermore , using inc, we have a diagonal map $\delta_A : A \longrightarrow A \otimes A$ in each fibre.

The thinking behind the definition is as follows. The category $\mathcal{C}$ with its finite product structure allows us to model an environment as the product of the types it contains. In the indexed category, program phrases defined in an environment will be modelled as elements in the fibre over the denotation of that environment.

$$[\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!]$$

$$\begin{array}{c} \mathbf{1} \\ \downarrow{\scriptstyle [\![x_1:\sigma_1,\ldots,x_n:\sigma_n \vdash M:\tau]\!]} \\ [\![\tau]\!] \end{array}$$

The isomorphism of adjunction κ is a first order binding construct that allows us to make the dependency of a program phrase on certain variables explicit. The negation functor is much as before, except that it now acts on those variables explicitly singled out by a previous κ.

$$\Gamma \times C \qquad \Gamma \qquad\qquad \Gamma \qquad \Gamma$$

$$\begin{array}{ccc} A & & A \times C \\ \downarrow & & \downarrow \\ B & & B \end{array} \qquad\qquad \begin{array}{ccc} A & & \neg B \\ \downarrow & & \uparrow \\ B & & \neg A \end{array}$$

The motivation for the axioms is as for $\otimes\neg$-categories, except that here, we can avoid one of the axioms as it follows from the indexing of $\neg$. However, we need our last axiom here in order to make the indexing of $\neg$ coherent: intuitively, it means that negating the retrieving of a value of type C from the environment to cons a value of type C to a value of type A gives us an operation of partially satisfying demand for a value of type C while leaving the demand for a value of type A untouched.

This formalism, unlike that for a $\otimes\neg$-category, separates the data and the control mechanisms. The indexed functor $\neg$ is in some sense oblivious to the indexed structure with which first order data manipulation is described. We do not want control to interfere with any data with which it is not concerned. So the ability to model continuations with indexed categories as we do here is a clear indication that we have separated the two. In the final section, we show that this modelling is essentially equivalent to that using premonoidal categories and self-adjointness. We take this as evidence that modelling continuations by self-adjointness is a robust notion in the sense that it is not overly sensitive to the way we model environments, as we could model them in two different ways, in each case fitting the self-adjointness into this framework.

To model λ+`callcc`, types are interpreted as objects in $\mathcal{C}$. Environments are interpreted using the product in $\mathcal{C}$.

$$\begin{aligned} [\![\neg\tau]\!] &\stackrel{\text{def}}{=} \neg[\![\tau]\!] \\ [\![\sigma \to \tau]\!] &\stackrel{\text{def}}{=} \neg\big([\![\sigma]\!] \otimes \neg[\![\tau]\!]\big) \\ [\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] &\stackrel{\text{def}}{=} [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \end{aligned}$$

A judgement $\Gamma \vdash M : \tau$ denotes an element $[\![\Gamma \vdash M : \tau]\!] : \mathbf{1} \longrightarrow [\![\tau]\!]$ in the fibre over $[\![\Gamma]\!]$.

$$
\begin{aligned}
[\![x_1 : \tau_1, \ldots x_n : \tau_n \vdash x_j : \tau_j]\!] &\stackrel{\text{def}}{=} H_{\pi_j}\kappa^{-1}(\mathsf{id}_{[\![\tau_j]\!]}) \\
[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] &\stackrel{\text{def}}{=} \mathsf{thunk}; \neg(\kappa\neg[\![\Gamma, x : \sigma \vdash M : \tau]\!]) \\
[\![\Gamma \vdash \mathtt{throw}\ M\ N : \sigma]\!] &\stackrel{\text{def}}{=} [\![\Gamma \vdash M : \neg\tau]\!]; \neg(\kappa(H_{\pi_1}[\![\Gamma \vdash N : \tau]\!])); \mathsf{force} \\
[\![\Gamma \vdash \mathtt{callcc}\ M : \tau]\!] &\stackrel{\text{def}}{=} [\![\Gamma \vdash M : \neg\tau \to \tau]\!]; \neg\delta; \mathsf{force} \\
[\![\Gamma \vdash MN : \tau]\!] &\stackrel{\text{def}}{=} [\![\Gamma \vdash M : \sigma \to \tau]\!]; \neg([\![\Gamma \vdash N : \sigma]\!] \otimes \neg[\![\tau]\!]); \mathsf{force}
\end{aligned}
$$

Again, the semantics as such is not the topic of the present paper. We only give some hint at how it is intended to work.

We write a morphism from X to Y in the fibre over C as

$$X \xrightarrow[C]{} Y$$

The most interesting clause is the one for λ-abstraction in that abstracting over a variable implies moving from one fibre to another.

In a more traditional (call-by-name) setting, λ would be interpreted by means of an adjoint to reindexing. Here, it is more elaborate, as it is decomposed into the first-order abstraction given by the structure on the fibration on the one hand and the fibrewise "negation" given by the continuation functor on the other.

A judgement $\Gamma, x : \sigma \vdash M : \tau$ denotes a morphism

$$\mathbf{1} \xrightarrow[[\![\star]\!]\times[\![\sigma]\!]]{} [\![\tau]\!]$$

Negating this given an arrow

$$\neg[\![\tau]\!] \xrightarrow[[\![\star]\!]\times[\![\sigma]\!]]{} \neg\mathbf{1}$$

which, by virtue of κ, amounts to

$$[\![\sigma]\!] \times \neg[\![\tau]\!] \xrightarrow[[\![\star]\!]]{} \neg\mathbf{1}$$

Negating this yields a morphism

$$\neg\neg\mathbf{1} \xrightarrow[[\![\star]\!]]{} \neg([\![\sigma]\!] \times \neg[\![\tau]\!])$$

All that remains to be done in order to get the meaning of $\lambda x.M$ is to precompose with $\mathsf{thunk} : \mathbf{1} \longrightarrow \neg\neg\mathbf{1}$, taking care of the double negation:

$$\mathbf{1} \xrightarrow[[\![\star]\!]]{} \neg\neg\mathbf{1} \xrightarrow[[\![\star]\!]]{} \neg([\![\sigma]\!] \times \neg[\![\tau]\!])$$

8 Relating $\otimes\neg$-categories and indexed $\neg$-categories

In this final section of the paper, we build upon the equivalence between κ-categories and symmetric premonoidal categories with the extra structure specified in Theorem 14 to relate $\otimes\neg$-categories and indexed $\neg$-categories. They are almost but not quite equivalent. The only difference lies implicit in Theorem 14: for our definition of $\otimes\neg$-category, we assert that the centre of our category has finite products, whereas Theorem 14 merely asserts that we have a category with finite products mapping, as the identity on objects, into the centre of our category. We regard this as a minor difference, as the latter merely extends the former mildly without changing any other structure. In fact, the latter concept seems category theoretically more natural, as explained in [15]. The concepts of $\otimes\neg$-category and indexed $\neg$-category are otherwise equivalent.

Let $\delta_A \stackrel{\text{def}}{=} \langle \mathsf{id}_A, \mathsf{id}_A \rangle : A \longrightarrow A \otimes A$

Let $\mathcal{K}$ be a $\otimes\neg$-category. Let $*_A$ be the Kleisli composition

$$f *_A g \stackrel{\text{def}}{=} \langle \pi_1, \mathsf{id} \rangle; A \otimes f \otimes g$$

Define $\neg_A : \mathbf{Kleisli}_{\mathcal{K}}(A \otimes (\_))^{\mathrm{op}} \longrightarrow \mathbf{Kleisli}_{\mathsf{Z}(\mathcal{K})}(A \times (\_))$ by $\neg_A B = B$ on objects and by

$$\neg_A f \stackrel{\text{def}}{=} A \otimes \neg f; \underline{\underline{\mathsf{apply}}}_A$$

on morphisms. This is well-defined: $A \otimes \neg f$ is central, because $\neg f$ is, and

$$\underline{\underline{\mathsf{apply}}} = \underline{\underline{\mathsf{apply}}}; \mathsf{thunk}; \neg\mathsf{thunk} = \mathsf{thunk}; \neg\neg\underline{\underline{\mathsf{apply}}}; \neg\mathsf{thunk}$$

is also central.

Define $\mathsf{force}_A : \neg_A \neg_A B \longrightarrow B$ in $\mathbf{Kleisli}_{\mathcal{K}}(A \otimes (\_))$ by $\mathsf{force}_A \stackrel{\text{def}}{=} \pi_2; \mathsf{apply}_{\mathbf{1}}$.

$\neg_A$ preserves identities $\pi_2 : A \otimes B \longrightarrow B$ because

$$\begin{aligned}
&\neg_A(\pi_2) \\
&= \neg_A(! \otimes B) \\
&= A \otimes \neg(! \otimes B); A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}); \mathsf{apply}_A \\
&= A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}; ! \otimes B); \mathsf{apply}_A \\
&= A \otimes \neg(! \otimes \neg\neg B; \mathbf{1} \otimes \mathsf{apply}_{\mathbf{1}}); \mathsf{apply}_A \\
&= A \otimes \neg(\mathbf{1} \otimes \mathsf{apply}_{\mathbf{1}}); A \otimes \neg(! \otimes \neg\neg B); \mathsf{apply}_A \\
&= A \otimes \neg(\mathbf{1} \otimes \mathsf{apply}_{\mathbf{1}}); ! \otimes \neg(\mathbf{1} \otimes \neg\neg B); \mathsf{apply}_{\mathbf{1}} \\
&= A \otimes \neg(\mathsf{apply}_{\mathbf{1}}); ! \otimes \neg\neg\neg B; \mathsf{apply}_{\mathbf{1}} \\
&= ! \otimes \neg B; \mathbf{1} \otimes \neg(\mathsf{apply}_{\mathbf{1}}); \mathsf{apply}_{\mathbf{1}} \\
&= ! \otimes \neg B \\
&= \pi_2 : A \otimes \neg B \longrightarrow \neg B
\end{aligned}$$

$\neg_A$ preserves composition: let $f : A \otimes B \longrightarrow C$ and $g : A \otimes C \longrightarrow D$. Then

$$\neg_A(f *_A g)$$

$$
\begin{aligned}
&= \neg_A(\delta_A \otimes B; A \otimes f; g) \\
&= A \otimes \neg(\delta_A \otimes B; A \otimes f; g); A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}); \mathsf{apply}_A \\
&= A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}; \delta_A \otimes B; A \otimes f; g); \mathsf{apply}_A \\
&= \ldots \\
&= \delta_A \otimes A \otimes A \otimes \neg C; A \otimes A \otimes \neg g; A \otimes A \otimes \neg(A \otimes f); \underline{\underline{\mathsf{apply}}}_{A \otimes A}
\end{aligned}
$$

$$
\begin{aligned}
&\neg_A(f) *_A \neg_A(g) \\
&= \delta_A \otimes \neg B; A \otimes A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}; f); A \otimes \mathsf{apply}_A; A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}; g); \mathsf{apply}_A \\
&= \ldots \\
&= \delta_A \otimes A \otimes A \otimes \neg C; A \otimes A \otimes \neg g; A \otimes A \otimes \neg(A \otimes f); A \otimes \underline{\underline{\mathsf{apply}}}_A; \underline{\underline{\mathsf{apply}}}_A
\end{aligned}
$$

So the required identity follows from the axiom

$$\mathsf{apply}_{A \otimes A'} = \langle \pi_2, \pi_1 \rangle \otimes \neg(A \otimes A' \otimes \neg B); A' \otimes \underline{\underline{\mathsf{apply}}}_A; \mathsf{apply}_{A'}$$

and the facts that $\langle \pi_2, \pi_1 \rangle$ is central and $\delta; \langle \pi_2, \pi_1 \rangle = \delta$.

The triangular identity $\neg_A \mathsf{force}_A *_A \mathsf{force}_A = \mathsf{id}$ holds:

$$
\begin{aligned}
&\neg_A \mathsf{force}_A *_A \mathsf{force}_A \\
&= \neg_A(\pi_2; \mathsf{apply}_{\mathbf{1}}) *_A \mathsf{force}_A \\
&= (A \otimes \neg \mathsf{apply}_{\mathbf{1}}; \neg_A(\pi_2)) *_A \mathsf{force}_A \\
&= (A \otimes \neg \mathsf{apply}_{\mathbf{1}}; \pi_2) *_A \mathsf{force}_A \\
&= \delta_A \otimes \neg B; A \otimes A \otimes \neg \mathsf{apply}_{\mathbf{1}}; A \otimes \pi_2; \pi_2; \mathsf{apply}_{\mathbf{1}} \\
&= A \otimes \neg \mathsf{apply}_{\mathbf{1}}; \delta_A \otimes \neg\neg\neg B; A \otimes \pi_2; \pi_2; \mathsf{apply}_{\mathbf{1}} \\
&= A \otimes \neg \mathsf{apply}_{\mathbf{1}}; \pi_2; \mathsf{apply}_{\mathbf{1}} \\
&= A \otimes \neg \mathsf{apply}_{\mathbf{1}}; ! \otimes \neg\neg\neg B; \mathsf{apply}_{\mathbf{1}} \\
&= ! \otimes \neg B; \mathbf{1} \otimes \neg \mathsf{apply}_{\mathbf{1}}; \mathsf{apply}_{\mathbf{1}} \\
&= \pi_2; \neg \mathsf{apply}_{\mathbf{1}}; \mathsf{apply}_{\mathbf{1}} \\
&= \pi_2 : A \otimes \neg B \longrightarrow \neg B
\end{aligned}
$$

force is natural:

$$
\begin{aligned}
&\neg_A \neg_A f *_A \mathsf{force}_A \\
&= A \otimes \neg(A \otimes \neg f; \underline{\underline{\mathsf{apply}}}_A); \underline{\underline{\mathsf{apply}}}_A; \mathsf{apply}_{\mathbf{1}} \\
&= A \otimes \neg \underline{\underline{\mathsf{apply}}}_A; A \otimes \neg(A \otimes \neg f); \mathsf{apply}_A \\
&= A \otimes \neg \underline{\underline{\mathsf{apply}}}_A; \mathsf{apply}_A; f \\
&= A \otimes \mathsf{apply}_{\mathbf{1}}; f \\
&= \mathsf{force}_A *_A f
\end{aligned}
$$

Putting this all together, it follows that we have

Proposition 18. *Given a $\otimes\neg$-category, $(\mathcal{K}, \neg, \mathsf{apply}, \mathsf{thunk})$, the construction $(\kappa(J), \neg_A, \mathsf{force}_A)$ as above, together with the given* thunk*, give an indexed $\neg$-category.*

Proof. Most of the proof is given above. For the rest, the axioms hold simply because the category $H_{\mathbf{1}}$ is given by $\mathcal{K}$. □

Theorem 19. *Given a symmetric premonoidal category $\mathcal{K}$ for which the premonoidal structure restricts to finite product structure on the centre, to extend this to the structure of a $\otimes\neg$-category is equivalent to extending the structure of the κ-category $\kappa(J)$ to that of an indexed $\neg$-category.*

Proof. We need to prove that the construction of the proposition is a bijection up to isomorphism. Given an indexed $\neg$-category, one can obtain a $\otimes\neg$-category by considering $H_{\mathbf{1}}$. In order to show that the construction applied to that $\otimes\neg$-category yields the original indexed $\neg$-category, everything is routine provided one can show that for any indexed $\neg$-category, the behaviour of $\neg$ on $H_{\mathbf{1}}$ determines its behaviour on H_A for all A. But this follows from the fact that $\neg$ is indexed and from the final axiom. □

There is little difference between the notions of indexed $\neg$-category and $\otimes\neg$-category. The only difference between them lies in the choice of an explicitly given category with finite products and an identity on objects strict monoidal functor into a symmetric premonoidal category rather than consideration of a property of the centre. The former is the structure given naturally by an indexed not-category. But also, that structure is a category theoretically more natural structure than that of $\otimes\neg$-category,, as explained in [15]. Computationally, however, it is natural to assume that in the presence of first-class continuations the whole of the centre admits finite products. This is becasue the self-adjoint structure allows every central morphism to be reified, as explained in [19].

9 Conclusions

We have shown how to account for environments in languages with a more computational flavour than the pure λ-calculus by generalising two semantics for environments and showing them essentially equivalent.

Modelling environments in some way is pervasive in semantics; however we have put here particular emphasis on continuation semantics, for the following reasons:

- First-class continuations allow the full `callcc` to be added to the language, which is the most powerful version of control found in actual languages. This contrasts with the situation for state, where only a rather weak notion of global state is added by commonly used notions like the state monad.
- The construct to be studied has universal properties on the category of computations. That does not seem to be the case for constructs associated with state, such as assignment.

– Continuations are an advanced concept in programming languages that could be made easier to use by semantic clarification. While local state has subtleties, it is not obvious if *global* variables as introduced by the state monad are all that much in need of elucidation.

(We have made a comparison with state here, as state and control appear to be the most natural things to add to a programming language, but this discussion would apply to other effects, such as exceptions.)

References

1. R.F. Blute, J.R.B. Cockett and R.A.G. Seely, Categories for computation in context and unified logic (submitted)
2. Gerard Boudol, The π-calculus in direct style, *Proc. POPL '97* .
3. Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Proc. ACM Symp. Principles of Programming Languages*, pages 163–173, January 1991.
4. Matthias Felleisen and Daniel P. Friedman. *The Seasoned Schemer.* MIT Press, 1996.
5. Andrzej Filinski, Declarative continuations: an investigation of duality in programming language semantics, *Proc. CTCS*, Lect. Notes in Computer Science 389 (1989) 224-249.
6. Michael Fourman and Hayo Thielecke. A proposed categorical semantics for ML modules. In David Pitt et al., editor, *Category Theory in Computer Science*, number 953 in LNCS. Springer Verlag, 1995.
7. Masahito Hasegawa, Decomposing typed lambda calculus into a couple of categorical programming languages, *Proc. CTCS*, Lect. Notes in Computer Science 953 (1995).
8. Bart Jacobs. *Categorical Type Theory.* PhD thesis, University of Nijmegen, 1991.
9. Richard Kiebutz, Borislav Agapiev, and James Hook. Three monads for continuations. In *Proceedings of the ACM Workshop on Continuations*, pages 39–48, 1992.
10. E. Moggi, Notions of computation and monads, Information and Computation 93 (1991) 55-92.
11. C.-H. L. Ong. A semantic view of classical proofs: type-theoretic, categorical, denotational characterizations. In *Proc. 11th IEEE Annual Symposium on Logic in Computer Science*, pages 230–241. IEEE Computer Society Press, 1996.
12. Robert Harper, Bruce Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4), October 1993.
13. P. Z. Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Comm. A.C.M.*, 4(1):55–58, January 1961.
14. Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings International Conference on Logic Programming and Automated Deduction*, number 624 in LNCS, pages 190–201, 1992.
15. A.J. Power, Premonoidal categories as categories with algebraic structure (submitted).

16. A.J. Power and E.P. Robinson, Premonoidal categories and notions of computation, *Proc. LDPL '96*, Math Structures in Computer Science (to appear).
17. E.P. Robinson and G. Rosolini, Categories of partial maps, Information and Computation 79 (1988) 95-130.
18. Hayo Thielecke. Continuation passing style and self-adjointness. In *Proceedings 2nd ACM SIGPLAN Workshop on Continuations*, number NS-96-13 in BRICS Notes Series, December 1996.
19. Hayo Thielecke, Continuations semantics and self-adjointness, *Proc. 13th MFPS*, Electronic Notes in Theoretical Computer Science 6 (to appear).
20. Hayo Thielecke, Categorical structure of continuation passing style, Edinburgh Ph.D. thesis (submitted).