Electronic Notes in Theoretical Computer Science 6 (1997)
URL: http://www.elsevier.nl/locate/entcs/volume6.html 17 pages

Continuation Semantics and Self-adjointness

Hayo Thielecke

LFCS, Department of Computer Science
University of Edinburgh
Edinburgh, Scotland, UK

Abstract

We give an abstract categorical presentation of continuation semantics by taking the continuation type constructor $\neg$ (or `cont` in Standard ML of New Jersey) as primitive. This constructor on types extends to a contravariant functor on terms which is adjoint to itself on the left; restricted to the subcategory of those programs that do not manipulate the current continuation, it is adjoint to itself on the right.

The motivating example of such a category is built from (equivalence classes of typing judgements for) continuation passing style (CPS) terms.

A call-by-value λ-calculus with the control operator `callcc` as well as a call-by-name λ-calculus can be interpreted. Arrow types are broken down into continuation types for argument/result-continuations pairs. Specialising the semantics to the CPS term model allows a reconstruction of CPS transforms.

1 Introduction

The task of finding a semantic infrastructure for continuation semantics is somewhat analogous to that of interpreting λ-calculus in a cartesian closed category. We need a first-order structure for interpreting environments and tuple types, in analogy with, but weaker than, cartesian products, as well as higher-order structure for interpreting continuation types. These now become the fundamental notion, while arrow types are derived as a special instance of continuation types. But whereas in λ-calculus every morphism is a "pure function", in CPS there is a need to identify a subcategory of effect-free computations (or values) that satisfy stronger properties than the general, possibly effectful, computations.

We show that effect-freeness in the presence of *first-class* continuations is a more subtle notion than would at first appear. In particular, it is not enough to exclude straightforward jumps like `throw k 42`.

In our framework, environments are modelled by means of a premonoidal category [9]: this is a categorical framework which provides enough parallelism on types to accommodate programs of multiple arity, but no real parallelism

on programs. For each object (type) A, there are functors $A \otimes (\_)$ and $(\_) \otimes A$. For morphisms $f : A \longrightarrow A'$ and $g : B \longrightarrow B'$, there are in general two *distinct* parallel compositions, $f \otimes B; A' \otimes g$ and $A \otimes g; f \otimes B'$. The *central* morphisms are those f such that for all g, the above composites agree. That is, those programs phrases which are not sensitive as to whether they are evaluated before or after any other. This provides a robust notion of effect-free morphism.

The continuation type constructor extends to a contravariant functor, as every function $\sigma \to \tau$ gives rise to a continuation transformer τ `cont` $\to \sigma$ `cont` in the opposite direction.

Categorically, the continuation type $\neg$ is introduced as a contravariant functor. This is adjoint to itself on the left, i.e.,

$$\frac{\neg B \longrightarrow C}{\neg C \longrightarrow B}$$

Intuitively, a morphism $\neg B \longrightarrow C$ expects both a B- and a C-accepting continuation as its argument and current continuation, respectively. The above correspondence arises by simply switching these. In Standard ML of New Jersey, we can define the unit `force` of this adjunction, the isomorphism of adjunction ϕ and the negation functor itself.

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

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

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

We require this to hold even "parametrically" in some other object A

$$\frac{A \otimes \neg B \longrightarrow C}{A \otimes \neg C \longrightarrow B}$$

The unit of this adjunction is the application map $\mathsf{apply} : A \otimes \neg(A \otimes \neg B) \longrightarrow B$. Among the central morphisms, $\neg$ is also adjoint to itself on the right.

$$\frac{A \longrightarrow \neg B}{B \longrightarrow \neg A}$$

Intuitively, a *central* morphism $A \longrightarrow \neg B$ expects an argument of type A and returns a B-accepting continuation. Hence there is demand for both A and B; and again the correspondence arises essentially by swapping.

The unit of this adjunction is a generic delaying map $\mathsf{thunk} : A \longrightarrow \neg\neg A$.

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

```
fun delay f x = ((negate(negate f)) o thunk) x;
    delay :  ('2a -> '2b) -> ('2a -> '2b cont cont);
```

Using thunk, we define a morphism

$$\mathsf{pair} : C \longrightarrow \neg(A \otimes \neg(A \otimes C))$$

which is a natural transformation in the centre. This in turn is used to define λ-abstraction.

$$\overline{\lambda}_A f \stackrel{\text{def}}{=} \mathsf{pair}; \neg(A \otimes \neg f)$$

This definition of λ-abstraction in terms of control (and tuple types) does not give rise to a closed category, although we have the following.

$$(A \otimes \overline{\lambda} f); \mathsf{apply} = A \otimes \mathsf{pair}; A \otimes \neg(A \otimes \neg f); \mathsf{apply} = A \otimes \mathsf{pair}; \mathsf{apply}; f = f$$

The corresponding $\overline{\lambda}(A \otimes g; \mathsf{apply}) = g$, however does not hold in general. Hence this λ-abstraction does not give rise to a cartesian closed category. But it is still sufficient for interpreting a call-by-value λ-calculus, as a central g can be pushed into λ (and values denote central morphisms). Although ML does not make this identification of function types $[A \to B]$ with $\neg(A \otimes \neg B)$, we can still define a pair of coercion functions:

```
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;
```

The motivating example of the categorical structure we shall discuss is a term model built form CPS terms. The objects of this category are type expressions and the morphisms are equivalence classes $[\vec{x}k \vdash M]$ of typing judgements $\vec{x}k \vdash M$ for CPS terms M in which a current continuation k has been singled out. For the CPS term model, the categorical semantics sketched above coincides with the corresponding CPS transforms.

Notational preliminaries

We let lowercase letters $x, y, n, m, k, l, \ldots$ range over variables (names) and uppercase letters $M, N, \ldots$ range over terms (in various calculi). $\vec{x}, \vec{y}, \ldots$ range over sequences $x_1 \ldots x_i$ of names. Commas in sequence are often omitted. When used as indices, lowercase letters range over natural numbers, e.g. $x_1 \ldots x_n$. We write $M[x \mapsto N]$ for the capture-avoiding substitution of N for x in M. Similarly, if $\vec{x} = x_1 \ldots x_j$ and $\vec{y} = y_1 \ldots y_j$, we write $M[\vec{x} \mapsto \vec{y}]$ for the simultaneous substitution of y_i for x_i $(i = 1, \ldots, j)$ in M. Categorical composition will also be written as $f; g \stackrel{\text{def}}{=} g \circ f$ (first f, then g).

2 Continuation-passing style

We consider the target language of CPS transforms as a calculus in its own right (similar to the intermediate language of the compiler in [1]). This is a (name-passing) calculus in which the only form of application is a "jump with arguments" [1] (for history see also [10],[11]).

2.1 CPS calculus

The BNF of CPS terms is as follows.

$$M ::= \quad x\langle\vec{x}\rangle \mid M\{x\langle\vec{x}\rangle \Leftarrow M\}$$

Here $k\langle\vec{x}\rangle$ is a jump to the continuation k with actual parameters $\vec{x}$, while $M\{n\langle\vec{x}\rangle = N\}$ binds the continuation with body N and formal parameters $\vec{x}$ to n in M.

We consider only simply typed CPS terms, that is those typeable according two these two rules:

$$\frac{}{\Gamma, k : \neg\vec{\tau}, \vec{y} : \vec{\tau} \vdash k\langle\vec{y}\rangle} \qquad \frac{\Gamma, n : \neg\vec{\tau} \vdash M \quad \Gamma, \vec{x} : \vec{\tau}, n : \neg\vec{\tau} \vdash N}{\Gamma \vdash M\{n\langle\vec{x}\rangle \Leftarrow N\}}$$

where $\tau ::= b \mid \neg\vec{\tau}$. Note that this gives a notion of recursion by means of a circular pointer when N contains its own "address" n. Free and bound variables are defined thus.

$$\begin{aligned} \mathsf{FV}(x\langle y_1 \ldots y_k\rangle) &= \{x, y_1, \ldots, y_k\} \\ \mathsf{FV}(M\{n\langle y_1 \ldots y_k\rangle \Leftarrow N\}) &= (\mathsf{FV}(M) \setminus \{n\}) \cup (\mathsf{FV}(N) \setminus \{n, y_1, \ldots, y_k\}) \end{aligned}$$

$$\begin{aligned} \mathsf{BV}(x\langle y_1 \ldots y_k\rangle) &= \emptyset \\ \mathsf{BV}(M\{n\langle y_1 \ldots y_k\rangle \Leftarrow N\}) &= \mathsf{BV}(M) \cup \mathsf{BV}(N) \cup \{n, y_1, \ldots, y_k\}) \end{aligned}$$

We adopt the usual convention that the sets of free and bound variables of a given term are disjoint, so as to prevent name capture. The axioms of the recursive CPS-calculus are as follows (with $n \neq m$).

$$L\{m\langle\vec{x}\rangle \Leftarrow M\}\{n\langle\vec{y}\rangle \Leftarrow N\} = L\{n\langle\vec{y}\rangle \Leftarrow N\}\{m\langle\vec{x}\rangle \Leftarrow M\{n\langle\vec{y}\rangle \Leftarrow N\}\} \quad \text{(Distr)}$$

$$k\langle\vec{y}\rangle\{n\langle\vec{z}\rangle \Leftarrow N\} = k\langle\vec{y}\rangle, \quad n \notin \mathsf{FV}(k\langle\vec{y}\rangle) \quad \text{(GC)}$$

$$n\langle\vec{y}\rangle\{n\langle\vec{z}\rangle \Leftarrow N\} = N[\vec{z} \mapsto \vec{y}]\{n\langle\vec{z}\rangle \Leftarrow N\} \quad \text{(RecJmp)}$$

$$M\{n\langle\vec{x}\rangle \Leftarrow m\langle\vec{x}\rangle\} = M[n \mapsto m] \quad \text{(RecEta)}$$

2.2 Translation from CPS calculus

The CPS calculus is a fragment (without mutual recursion) of the intermediate language used in [1].

CPS-calculus	Appel's datatype `cexp`
$x\langle y_1 \ldots y_j \rangle$	`APP(VAR` x`,[VAR` y_1`,...,VAR` y_j`])`
$M\{n\langle x_1 \ldots x_j \rangle \Leftarrow N\}$	`FIX([(`n`,[`x_1`,...,`x_j`],`N`)],`M`)`

CPS calculus can be translated to the simply-typed λ-calculus with a fixpoint operator as follows.

$$\begin{aligned} k\langle y_1 \ldots y_n \rangle^\circ &= k y_1 \ldots y_n \\ (M\{n\langle \vec{x} \rangle \Leftarrow N\})^\circ &= (\lambda n.M^\circ)(\mu n.\lambda \vec{x}.N^\circ) \end{aligned}$$

where μ is a fixpoint-finder in the simply-typed λ-calculus satisfying $\mu x.M = M[x \mapsto \mu x.M]$

Proposition 2.1 *The translation* $(\_)^\circ$ *is sound.*

For the non-recursive fragment of CPS calculus, one can simplify the translation to λ-calculus, not requiring the fixpoint combinator.

$$\begin{aligned} k\langle y_1 \ldots y_n \rangle^\circ &= k y_1 \ldots y_n \\ M\{n\langle y_1 \ldots y_n \rangle \Leftarrow N\}^\circ &= (\lambda n.M^\circ)(\lambda y_1 \ldots y_n.N^\circ) \end{aligned}$$

As CPS calculus is itself a polyadic name-passing calculus, it can be translated to the π-calculus quite easily. A jump $k\langle \vec{x} \rangle$ corresponds to a π-calculus output particle. The continuation binding construct is translated into Sangiorgi's "local environment idiom" [6].

$$\begin{aligned} k\langle x_1 \ldots x_i \rangle^\bullet &= \overline{k}\langle x_1 \ldots x_i \rangle \\ M\{n\langle x_1 \ldots x_i \rangle \Leftarrow N\}^\bullet &= (\nu n)(M^\bullet \mid !n(x_1 \ldots x_i).N^\bullet) \end{aligned}$$

2.3 CPS transforms

The typing for first-class continuations in Standard ML of New Jersey [2] is given by that of simply-typed λ-calculus and the two rules for the continuation primitives `callcc` and `throw`.

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

The canonical CPS transform is the one from [2]

Definition 2.2 *The call-by-value* CPS *transform for* λ*-calculus with* `callcc` *is defined inductively as follows, where* $(\!|M|\!)(k)$ *is the transform of the* λ*-term* M *with respect to the current continuation* k*.*

$$
\begin{aligned}
(\!|x|\!)(k) &= k\langle x\rangle \\
(\!|\lambda x.M|\!)(k) &= k\langle q\rangle\{q\langle xh\rangle \Leftarrow (\!|M|\!)(h)\} \\
(\!|\mu f.\lambda x.M|\!)(k) &= k\langle f\rangle\{f\langle xh\rangle \Leftarrow (\!|M|\!)(h)\} \\
(\!|MN|\!)(k) &= (\!|M|\!)(m)\{m\langle f\rangle \Leftarrow (\!|N|\!)(n)\{n\langle a\rangle \Leftarrow f\langle ak\rangle\}\} \\
(\!|\texttt{throw}\ M\ N|\!)(k) &= (\!|M|\!)(m)\{m\langle n\rangle \Leftarrow (\!|N|\!)(n)\} \\
(\!|\texttt{callcc}\ M|\!)(k) &= (\!|M|\!)(m)\{m\langle f\rangle \Leftarrow f\langle kk\rangle\}
\end{aligned}
$$

The translation extends to types:

$$
\begin{aligned}
(\!|\sigma \to \tau|\!) &= \neg((\!|\sigma|\!), \neg(\!|\tau|\!)) \\
(\!|\neg\tau|\!) &= \neg(\!|\tau|\!)
\end{aligned}
$$

We do not consider the control operators in the call-by-name semantics as their intended meaning is less clear than in the case of call-by-value.

Definition 2.3 *The lazy call-by-name* CPS *transform is defined as follows.*

$$
\begin{aligned}
\mathcal{L}(\!|x|\!)(k) &= x\langle k\rangle \\
\mathcal{L}(\!|\lambda x.M|\!)(k) &= k\langle f\rangle\{f\langle xh\rangle \Leftarrow \mathcal{L}(\!|M|\!)(h)\} \\
\mathcal{L}(\!|MN|\!)(k) &= \mathcal{L}(\!|M|\!)(m)\{m\langle f\rangle \Leftarrow f\langle nk\rangle\{n\langle h\rangle \Leftarrow \mathcal{L}(\!|N|\!)(h)\}\}
\end{aligned}
$$

Despite being traditionally called "call-by-name", this transform does not satisfy the η law, only (like the call-by-value one) the η_V law. This is because it is "lazy" in the sense that λ-abstraction delays the evaluation of the body (sometimes called "protecting by a λ"). That is why we qualify "call-by-name" with "lazy" to distinguish this transform from non-lazy alternatives that satisfy the full η law. (Unfortunately, "lazy" is sometimes used to mean call-by-need.) A genuinely call-by-name CPS transform not suffering from laziness can be given by uncurrying all function types. (In [4] a related transform is given. But as the target language is a λ-calculus, it is, apart from the thunking of base types, the identity.)

Definition 2.4 *The (uncurrying) call-by-name CPS transform is defined as follows*

$$
\begin{aligned}
\mathcal{N}(\!|x|\!)(k) &= k\langle x\rangle \\
\mathcal{N}(\!|\lambda x.M|\!)(k) &= k\langle f\rangle\{f\langle x\vec{y}h\rangle \Leftarrow \mathcal{N}(\!|M|\!)(m)\{m\langle g\rangle \Leftarrow g\langle \vec{y}h\rangle\}\} \\
\mathcal{N}(\!|MN|\!)(k) &= \mathcal{N}(\!|M|\!)(m)\{m\langle f\rangle \Leftarrow \mathcal{N}(\!|N|\!)(n)\{n\langle a\rangle \Leftarrow k\langle g\rangle\{g\langle \vec{y}k\rangle \Leftarrow f\langle a\vec{y}k\rangle\}\}\}
\end{aligned}
$$

Function types are translated by being uncurried:

$$\mathcal{N}(\!|\tau_1 \to \cdots \to \tau_n \to b|\!) = \neg(\mathcal{N}(\!|\tau_1|\!), \ldots, \mathcal{N}(\!|\tau_n|\!)\neg b)$$

3 Categories for continuation semantics

The basis for our categorical account of continuation semantics will be the negation functor, corresponding to the typing based on $\neg$ in section 2.1. However, the continuations considered there were actually *polyadic*, that is, in $k\langle x_1 \ldots x_i\rangle$ k is applied to a tuple of arguments. That is why, before introducing $\neg$, we need some first-order structure for building up such tuples (as well

as environments).

3.1 Premonoidal as first-order structure

We will use a premonoidal structure $\otimes$ for interpreting environments

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!]$$

as $[\![\tau_1]\!] \otimes \cdots \otimes [\![\tau_n]\!]$. (What we give here is the special case of a *strict* premonoidal category, largely because this is easier to present without going into details of coherence. As the leading example is a term model, we can get away with assuming strictness.)

Definition 3.1 ([9]) *A strict premonoidal category is a category $\mathcal{K}$ together with an object $\mathbf{1} \in \mathbf{Ob}\mathcal{K}$ and, for each $A \in \mathbf{Ob}\mathcal{K}$, endofunctors $A \otimes (\_)$ and $(\_) \otimes A$ that agree in the sense that*

$$(A \otimes (\_))(B) = ((\_) \otimes B))(A) =: A \otimes B$$

such that $\mathbf{1} \otimes (-) = \mathsf{id}_{\mathcal{K}} = (-) \otimes \mathbf{1}$ and

$$(A \otimes B) \otimes C = A \otimes (B \otimes C)$$
$$(f \otimes B) \otimes C = f \otimes (B \otimes C)$$
$$(A \otimes g) \otimes C = A \otimes (g \otimes C)$$
$$(A \otimes B) \otimes h = A \otimes (B \otimes h)$$

A morphism $f : A \longrightarrow A'$ is called central if it commutes with everything in the sense that, for all $g : B \longrightarrow B'$, we have

$$A \otimes g; f \otimes B' = f \otimes B; A' \otimes g$$
$$g \otimes A; B' \otimes f = B \otimes f; g \otimes A'$$

The centre $\mathsf{Z}(\mathcal{K})$ of $\mathcal{K}$ is the subcategory of $\mathcal{K}$ consisting of all objects and all central morphisms. Let $\iota : \mathbf{Z}(\mathcal{K}) \hookrightarrow \mathcal{K}$ be the inclusion.

Definition 3.2 *A $\otimes$-category is a strict premonoidal category $\mathcal{K}$ such that $\otimes$ is a cartesian product in the centre of $\mathcal{K}$ and furthermore, the twist map arising from this product*

$$\langle \pi_2, \pi_1 \rangle : A \otimes B \longrightarrow B \otimes A$$

is natural in A and and in B.

We extend the morphism pairing operation $\langle \_, \_ \rangle$, given by the product in the centre, to all of $\mathcal{K}$ as follows. For $f : C \longrightarrow A$ and $g : C \longrightarrow B$, let

$$\langle f, g \rangle \stackrel{\text{def}}{=} \langle \mathsf{id}_C, \mathsf{id}_C \rangle; C \otimes g; f \otimes B \quad : \quad C \longrightarrow A \otimes B$$

3.2 Continuation types as higher-order structure

We will be interested in a particularly simple kind of adjunction: a contravariant functor being adjoint to its own dual, with the unit and co-unit being the

same.

Definition 3.3 *A functor $F : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}$ is called* self-adjoint on the left *iff there is a natural transformation $\epsilon : FF^{\mathrm{op}} \longrightarrow \mathsf{id}_{\mathcal{C}}$ such that $F\epsilon; \epsilon F = id$. Dually, F is called* self-adjoint on the right *iff F^{op} is self-adjoint on the left.*

The continuation functor $\neg$ has two universal properties, adjointness on the left and right; we axiomatise them here in terms of the universal maps apply and thunk, respectively.

Definition 3.4 *A $\otimes\neg$-category is a $\otimes$-category $\mathcal{K}$ together with*

- *a functor $\neg : \mathcal{K}^{\mathrm{op}} \longrightarrow \mathsf{Z}(\mathcal{K})$ such that for each object A of $\mathcal{K}$, $(A \otimes \iota\neg(\_)) : \mathcal{K}^{\mathrm{op}} \longrightarrow \mathcal{K}$ is self-adjoint on the left (let $\mathsf{apply}_A : A \otimes \neg(A \otimes \neg B) \longrightarrow B$ be its unit), and*
- *a natural transformation $\mathsf{thunk} : \mathsf{id}_{\mathsf{Z}(\mathcal{K})} \longrightarrow \neg\neg$ in $\mathsf{Z}(\mathcal{K})$ such that $\mathsf{thunk}; \mathsf{force} = \mathsf{id}$ where $\mathsf{force} \stackrel{\mathrm{def}}{=} \mathsf{apply}_{\mathbf{1}} : \neg\neg A \longrightarrow A$*

such that

- apply *is dinatural in A and*
- *letting $\underline{\underline{\mathsf{apply}}}_A \stackrel{\mathrm{def}}{=} A \otimes \neg(A \otimes \mathsf{force}); \mathsf{apply}_A$, we have*

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

The first of these four axioms establishes another link between forcing and thunking (in addition to the more familiar $\mathsf{thunk}; \mathsf{force} = \mathsf{id}$); the second states that the call-by-name application, unlike the call-by-value one, is effect-free; the other two are somewhat technical coherence conditions.

Remark 3.5 *What is perhaps surprising about this definition are the strong assumptions we have made about the centre. All central morphisms are deemed to be effect-free, so that they respect the product. While centrality is certainly necessary for effect-freeness, there is in general no reason to assume that it is sufficient. It appears to be the presence of first-class continuations, specifically the unit* force*, that that makes centrality such a strong property: if a morphism commutes with everything, it must commute with* force*, and that implies that it commutes with reification. Slightly more technically, if $f : A \longrightarrow B$ is central, then $f \otimes \neg\neg\neg B; B \otimes \mathsf{force} = A \otimes \mathsf{force}; f \otimes \neg B$. This implies the naturality of* thunk*, as*

$$\begin{aligned} & f; \mathsf{thunk} \\ &= A \otimes \mathsf{thunk}; A \otimes \neg(f \otimes \neg\neg\neg B; B \otimes \mathsf{force}; \underline{\underline{\mathsf{apply}}}); \mathsf{apply} \\ &= A \otimes \mathsf{thunk}; A \otimes \neg(A \otimes \mathsf{force}; f \otimes \neg B; \underline{\underline{\mathsf{apply}}}); \mathsf{apply} \\ &= \mathsf{thunk}; \neg\neg f \end{aligned}$$

Remark 3.6 *Instead of defining $\neg$ to have the centre as it codomain, we could have required the adjoint correspondence*

$$\frac{A \otimes \neg B \longrightarrow C}{A \otimes \neg C \longrightarrow B}$$

to be natural in A , as this implies that every morphism of the form $\neg f$ is central. This property is perhaps more intuitive in terms of control flow: control manipulation concerning B and C does not affect a separate strand of control $g : A \longrightarrow A'$.

Remark 3.7 *Given a cartesian closed category $\mathcal{C}$ (with strict products), and an object R of $\mathcal{C}$ we can define a $\otimes\neg$-category $\mathcal{K}$ as follows*

$$\begin{aligned} \mathbf{Ob}\mathcal{K} &\stackrel{\text{def}}{=} \mathbf{Ob}\mathcal{C} \\ \mathcal{K}(A,B) &\stackrel{\text{def}}{=} \mathcal{C}(R^B, R^A) \end{aligned}$$

$A \otimes (\_)$ is given by the product $A \times (\_)$ in $\mathcal{C}$. The functor $\neg$ is $R^{(\_)}$. $\mathsf{force} \stackrel{\text{def}}{=} R^{\eta}$ and $\mathsf{thunk} \stackrel{\text{def}}{=} \eta R^{(\_)}$, where $\eta_A : A \longrightarrow R^{R^A}$ is the unit of the "continuation monad" on $\mathcal{C}$.

Despite the apparent generality of this construction, we regard this as an overly specific approach that does not do justice to the full generality of CPS (compare section 2.2). It consists essentially of implementing CPS in simply-typed λ-calculus and then interpreting this in the usual fashion in a cartesian closed category.

3.3 λ-abstraction in a $\otimes\neg$-category

Just as in the standard CPS transforms, function types $\sigma \to \tau$ will be decomposed into continuations for arguments σ and result continuations $\neg\tau$. So instead of exponentials, we have a derived notion of arrow type

$$[A \to B] \stackrel{\text{def}}{=} \neg(A \otimes \neg B)$$

The corresponding application map is the unit of the adjunction

$$\mathsf{apply}_A : A \otimes \neg(A \otimes \neg B) \longrightarrow B$$

This can be decomposed into two steps: first, the argument of type A is supplied, giving a thunk of type $\neg\neg B$, which no longer depends on an A; and this is then evaluated (forced) to give the result of type B.

$$\mathsf{apply} = \underline{\underline{\mathsf{apply}}}; \mathsf{force}$$

For call-by-name, only the argument is supplied, while the result remains unevaluated.

$$\underline{\underline{\mathsf{apply}}}_A \stackrel{\text{def}}{=} A \otimes \neg(A \otimes \mathsf{force}); \mathsf{apply} \quad : A \otimes \neg(A \otimes B) \longrightarrow \neg B$$

In a cartesian closed category, we could define λ-abstraction in terms of the right adjoint $[A \to (\_)]$ in $A \times (\_) \dashv [A \to (\_)]$ and the unit of adjunction (curried pairing map) $\mathsf{pair} : C \longrightarrow [A \longrightarrow (A \times C)]$ as $\lambda f \stackrel{\text{def}}{=} \mathsf{pair}; [A \to f]$. The notion of λ-abstraction that we have in the present setting can be defined in a way that is formally very similar, although we do not have cartesian closure.

We define a pairing map as

$$\mathsf{pair} \stackrel{\text{def}}{=} \mathsf{thunk}_C; \neg\underline{\underline{\mathsf{apply}}}_A : C \longrightarrow \neg(A \otimes \neg(A \otimes C))$$

This then allows us to define (call-by-value) λ-abstraction.

$$\overline{\lambda} f \stackrel{\text{def}}{=} \mathsf{pair}; \neg(A \otimes \neg f)$$

Although we may read apply and pair as having the types familiar from cartesian closed categories, that is

$$\begin{aligned} \mathsf{apply} &: A \otimes [A \to B] \longrightarrow B \\ \mathsf{pair} &: C \longrightarrow [A \to (A \otimes B)] \end{aligned}$$

this is really a kind of secondary etymology, as in reality apply and pair are the unit/counit of negation functors $A \otimes \neg(\_)$ and $\neg(A \otimes (\_))$, respectively.

$$\begin{aligned} \mathsf{apply} &: A \otimes \neg(A \otimes \neg B) \longrightarrow B \\ \mathsf{pair} &: C \longrightarrow \neg(A \otimes \neg(A \otimes C)) \end{aligned}$$

Although a $\otimes\neg$-category is not monoidally closed, it is centrally closed in the sense of [8].

Proposition 3.8 *$\iota((\_) \otimes A) \dashv \neg(A \otimes \neg(\_))$ where $\iota : \mathsf{Z}(\mathcal{K}) \hookrightarrow \mathcal{K}$ is the inclusion of the centre.*

It has already been mentioned that $\underline{\underline{\mathsf{apply}}}$ can be seen as a call-by-name application map that does not force the result. A corresponding call-by-name λ-abstraction can be defined in terms of $\overline{\lambda}$.

$$\underline{\underline{\lambda}}_A f \stackrel{\text{def}}{=} \overline{\lambda}_A f; \neg(A \otimes \mathsf{force})$$

Proposition 3.9 *$\underline{\underline{\lambda}}$ and $\underline{\underline{\mathsf{apply}}}$ satisfy the following equations:*

$$\begin{aligned} A \otimes \underline{\underline{\lambda}}_A f; \underline{\underline{\mathsf{apply}}}_A &= f \\ \underline{\underline{\lambda}}_A (A \otimes g; \underline{\underline{\mathsf{apply}}}_A) &= g \quad \textit{for central } g \end{aligned}$$

4 The CPS term model

In this section, the motivating example of the above categorical framework is constructed from the syntax of CPS calculus (section 2.1). We show that this forms a $\otimes\neg$-category and exhibit the connection between some notions of effect-free morphism.

Definition 4.1 *The category $\mathcal{K}(\mathrm{CPS})$ is constructed as follows. Objects are sequences $\vec{\tau}$ of types. A morphism from $\vec{\sigma}$ to $\vec{\tau}$ is an equivalence class*

$$[\vec{x}{:}\vec{\sigma}, k{:}\neg\vec{\tau} \vdash M]$$

of judgements, where $\vec{x}{:}\vec{\sigma}, k{:}\neg\vec{\tau} \vdash M$ and $\vec{x}'{:}\vec{\sigma}', k'{:}\neg\vec{\tau}' \vdash M'$ are equivalent iff

$$M = M'[\vec{x}'k' \mapsto \vec{x}k]$$

is derivable from the axioms of CPS calculus. $\otimes$ on objects is concatenation of sequences. The structure on morphisms is given as follows:

$$\begin{aligned}
\mathsf{id}_{\vec{\sigma}} &= [\vec{x}{:}\vec{\sigma}, k{:}\neg\vec{\sigma} \vdash k\langle\vec{x}\rangle] \\
[\vec{x}{:}\vec{\tau}_1, n{:}\neg\vec{\tau}_2 \vdash M]; [\vec{y}{:}\vec{\tau}_2, h{:}\neg\vec{\tau}_3 \vdash N] &= [\vec{x}{:}\vec{\tau}_1, h{:}\neg\vec{\tau}_3 \vdash M\{n\langle\vec{y}\rangle \Leftarrow N\}] \\
\neg[\vec{x}{:}\vec{\sigma}, h{:}\neg\vec{\tau} \vdash M] &= [h{:}\neg\vec{\tau}, k{:}\neg\neg\vec{\sigma} \vdash k\langle f\rangle\{f\langle\vec{x}\rangle \Leftarrow M\}] \\
[\vec{x}{:}\vec{\sigma}_1, k{:}\neg\tau \vdash M] \otimes \vec{\sigma}_2 &= [\vec{x}{:}\vec{\sigma}_1, \vec{y}{:}\vec{\sigma}_2, h{:}\neg(\vec{\tau}\vec{\sigma}_2) \vdash M\{k\langle\vec{z}\rangle \Leftarrow h\langle\vec{z}\vec{y}\rangle\}] \\
\vec{\sigma}_1 \otimes [\vec{x}{:}\vec{\sigma}_2, k{:}\neg\tau \vdash M] &= [\vec{y}{:}\vec{\sigma}_1, \vec{x}{:}\vec{\sigma}_1, h{:}\neg(\vec{\sigma}_1\vec{\tau}) \vdash M\{k\langle\vec{z}\rangle \Leftarrow h\langle\vec{y}\vec{z}\rangle\}] \\
\mathsf{apply}_{\vec{\sigma}} &= [\vec{x}{:}\vec{\sigma}, f{:}\neg(\vec{\sigma}, \neg\vec{\tau}), k{:}\neg\vec{\tau} \vdash f\langle\vec{x}k\rangle] \\
\mathsf{thunk}_{\vec{\tau}} &= [\vec{x}{:}\vec{\tau}, k{:}\neg\neg\neg\vec{\tau} \vdash k\langle f\rangle\{f\langle h\rangle \Leftarrow h\langle\vec{x}\rangle\}]
\end{aligned}$$

In the sequel, we will omit the types from the judgements when they are clear from the context, listing only the free variables, among them the current continuation. For instance we write $[\vec{x}k \vdash k\langle\vec{x}\rangle].$ for the identity. Some other important morphisms are the following.

$$\begin{aligned}
\mathsf{force} &= [fk \vdash f\langle k\rangle] \\
\mathsf{pair} &= [\vec{x}k \vdash k\langle f\rangle\{f\langle\vec{y}h\rangle \Leftarrow h\langle\vec{y}\vec{x}\rangle\}] \\
\overline{\lambda}[\vec{y}xk \vdash M] &= [\vec{y}h \vdash h\langle f\rangle\{f\langle xk\rangle \Leftarrow M\}] \\
\underline{\lambda}[\vec{y}xk \vdash M] &= [\vec{y}k \vdash k\langle f\rangle\{f\langle xk\rangle \Leftarrow M\{k\langle g\rangle \Leftarrow g\langle\vec{y}k\rangle\}\}] \\
\underline{\underline{\mathsf{apply}}} &= [xfk \vdash k\langle g\rangle\{g\langle\vec{y}k\rangle \Leftarrow f\langle x\vec{y}k\rangle\}]
\end{aligned}$$

Also note that the adjoint correspondent of a morphism $f = [hk \vdash M]$, given by $\phi f \stackrel{\text{def}}{=} \neg f; \mathsf{force}$, is just $[kh \vdash M]$.

Proposition 4.2 *Let $f = [\vec{x}k \vdash M]$. Then the following are equivalent*

(i) *f is central*

(ii) *for all N with $k, \vec{z} \notin \mathsf{FV}(N)$ and $h, \vec{w} \notin \mathsf{FV}(M)$*

$$M\{k\langle\vec{z}\rangle \Leftarrow N\{h\langle\vec{w}\rangle \Leftarrow l\langle\vec{z}\vec{w}\rangle\}\} = N\{h\langle\vec{w}\rangle \Leftarrow M\{k\langle\vec{z}\rangle \Leftarrow l\langle\vec{z}\vec{w}\rangle\}\}$$

(iii) $f; \mathsf{thunk} = \mathsf{thunk}; \neg\neg f$

Proof. We show only f central implies $f; \mathsf{thunk} = \mathsf{thunk}; \neg\neg f$, as this is the most crucial. (Compare remark 3.5 and figure 1.)

If $f = [\vec{x}k \vdash M]$ is central, then

$$M\{k\langle\vec{y}\rangle \Leftarrow h\langle f\rangle\{f\langle l\rangle \Leftarrow z\langle\vec{y}l\rangle\}\} = h\langle f\rangle\{f\langle l\rangle \Leftarrow M\{k\langle\vec{y}\rangle \Leftarrow z\langle\vec{y}l\rangle\}\}$$

Hence, applying $\_\{z\langle\vec{y}l\rangle \Leftarrow l\langle\vec{y}\rangle\}$, we have

$$\begin{aligned} & M\{k\langle\vec{y}\rangle \Leftarrow h\langle f\rangle\{f\langle l\rangle \Leftarrow z\langle\vec{y}l\rangle\}\}\{z\langle\vec{y}l\rangle \Leftarrow l\langle\vec{y}\rangle\} \\ = {} & h\langle f\rangle\{f\langle l\rangle \Leftarrow M\{k\langle\vec{y}\rangle \Leftarrow z\langle\vec{y}l\rangle\}\}\{z\langle\vec{y}l\rangle \Leftarrow l\langle\vec{y}\rangle\} \end{aligned}$$

And this simplifies to

$$M\{k\langle\vec{y}\rangle \Leftarrow h\langle f\rangle\{f\langle l\rangle \Leftarrow l\langle\vec{y}\rangle\}\} = h\langle f\rangle\{f\langle k\rangle \Leftarrow M\} \tag{1}$$

Now

$$\begin{aligned} & \neg\neg f \\ = {} & \neg\neg[\vec{x}k \vdash M] \\ = {} & \neg[kp \vdash p\langle g\rangle\{g\langle\vec{x}\rangle \Leftarrow M\}] \\ = {} & [ph \vdash h\langle f\rangle\{f\langle k\rangle \Leftarrow p\langle g\rangle\{g\langle\vec{x}\rangle \Leftarrow M\}\}] \end{aligned}$$

$$\begin{aligned} & \mathsf{thunk}; \neg\neg f \\ = {} & [\vec{x}n \vdash n\langle p\rangle\{p\langle k\rangle \Leftarrow k\langle\vec{x}\rangle\}]; [ph \vdash h\langle f\rangle\{f\langle k\rangle \Leftarrow p\langle g\rangle\{g\langle\vec{x}\rangle \Leftarrow M\}\}] \\ = {} & [\vec{x}h \vdash n\langle p\rangle\{p\langle k\rangle \Leftarrow k\langle\vec{x}\rangle\}\{n\langle p\rangle \Leftarrow h\langle f\rangle\{f\langle k\rangle \Leftarrow p\langle g\rangle\{g\langle\vec{x}\rangle \Leftarrow M\}\}\}] \\ = {} & [\vec{x}h \vdash h\langle f\rangle\{f\langle k\rangle \Leftarrow p\langle g\rangle\{g\langle\vec{x}\rangle \Leftarrow M\}\}\{p\langle k\rangle \Leftarrow k\langle\vec{x}\rangle\}] \\ = {} & [\vec{x}h \vdash h\langle f\rangle\{f\langle k\rangle \Leftarrow g\langle\vec{x}\rangle\{g\langle\vec{x}\rangle \Leftarrow M\}\}] \\ = {} & [\vec{x}h \vdash h\langle f\rangle\{f\langle k\rangle \Leftarrow M\}] \end{aligned}$$

$$\begin{aligned} & f; \mathsf{thunk} \\ = {} & [\vec{x}k \vdash M]; [\vec{y}h \vdash h\langle f\rangle\{f\langle l\rangle \Leftarrow l\langle\vec{y}\rangle\}] \\ = {} & [\vec{x}h \vdash M\{k\langle\vec{y}\rangle \Leftarrow h\langle f\rangle\{f\langle l\rangle \Leftarrow l\langle\vec{y}\rangle\}\}] \end{aligned}$$

So by (1), $\mathsf{thunk}; \neg\neg f = f; \mathsf{thunk}$. □

Proposition 4.3 *$\otimes$ is a product in the centre of $\mathcal{K}$(CPS).*

Proposition 4.4 *$\mathcal{K}$(CPS) is a $\otimes\neg$-category.*

The proof is by calculation with the axioms of CPS calculus. In particular, naturality and dinaturality seem to be closely related to the static binding of continuation names.

5 Effects and values in the presence of continuations

This section discusses the notion of effect-free program phrases in the presence of first-class continuations.

5.1 Copying and discarding

Those morphisms that are discardable in the sense of $f; !_B = !_A : A \longrightarrow 1$ (called total in [3]) are not sufficiently effect-free to be copyable.

Consider the following morphism in $\mathcal{K}$(CPS).

$$\mathsf{twicecc} \stackrel{\text{def}}{=} [\vec{x}hk \vdash k\langle\vec{x}l\rangle\{l\langle\vec{y}\rangle \Leftarrow k\langle\vec{y}h\rangle\}] : A \otimes \neg A \longrightarrow A \otimes \neg A$$

Informally, in terms of continuation transformers, this could be read as "$k \mapsto k \circ k$". This does not respect the product, in that copying the computation and

copying its "value" produce different results. Rather than give a complicated CPS calculation, we illustrate this with the following SML code. Using the familiar `fun twice f = f o f;` and the categorical combinators `thunk`, `phi`, ... above, we define an ML function `twicecc` as follows.

```
fun twicecc a = (phi(funtocont o twice o conttofun) o thunk) a;
    twicecc :  'a * 'a cont -> 'a * 'a cont;

fun copy_twicecc x = (twicecc x, twicecc x);


fun twicecc_copy x = (fn y => (y,y)) (twicecc x);

fun distinguisher testee =
    callcc(fn k =>
           (fn (((n,h),f),(_,g)) =>
            throw h (conttofun f (conttofun g  n)))
           (testee ((0,k),funtocont (fn n => n + 1))));

- distinguisher copy_twicecc;
val it = 3 : int
- distinguisher twicecc_copy;
val it = 4 : int
```

The context that can distinguish `copy_twicecc` and `twicecc_copy`, abstracted as `distinguisher` above, could be visualised as follows.

$$(\underbrace{I}_{0} \otimes \underbrace{\neg I}_{k}) \otimes \underbrace{\neg(I \otimes \neg I)}_{inc} \longrightarrow ((\underbrace{I}_{n} \otimes \underbrace{\neg I}_{h}) \otimes \underbrace{\neg(I \otimes \neg I)}_{f}) \otimes ((I \otimes \neg I) \otimes \underbrace{\neg(I \otimes \neg I)}_{g})$$

Using `twicecc` and `distinguisher`, we can define a context to show that force applied to some argument is not copyable either.

```
fun forcecopytester f =  distinguisher (f o (delay twicecc));
```

This distinguishes `(force h, force h)` and `(fn a => (a,a)) (force h)`. Hence copyability and discardability are orthogonal.

	copyable	not copyable
discardable	x $\lambda x.M$	`twicecc a`
not discardable	`throw k 42`	`force h`

5.2 Centrality and effect-freeness

We mentioned in remark 3.5 that it is due to the self-adjointness that centrality can be assumed to imply effect-freeness. There is some room for misunderstanding here, as there is a different, but weaker, argument for such an implication. We hope to clarify the connection between centrality and effect-freeness *in the presence of first-class continuations* by some concrete examples

```
fun forcefirst (a,b) =
    let val y = force b in
    let val x =  (output(std_out, "A side effect.\n"); 42) in
        (x,y) end end;

fun forcelast (a,b) =
    let val x =  (output(std_out, "A side effect.\n"); 42) in
    let val y = force b in
        (x,y) end end;

fun trytoreify f (n,k) =
    (phi(fn h  =>
         (fn (x,y) => throw y x)
         (f ((n,k),h)))) (thunk ());

val effectnotinclosure = trytoreify forcelast ((),());
    effectnotinclosure  : int cont cont;


val effectinclosure = trytoreify forcefirst ((),());
    effectinclosure  : int cont cont;

- force effectnotinclosure;
val it = 42 : int
- force effectinclosure;
A side effect.
val it = 42 : int
```

Fig. 1. force can reify a computation by being precomposed

First note that we can talk about centrality in quite a general setting: whenever we have a language having a `let` and a tuple construct, we can define a term M to be central iff for all fresh variables a and b and all other terms N,

$$\texttt{let}\ a = M\ \texttt{in}\ \texttt{let}\ b = N\ \texttt{in}\ (a, b)$$

is the same (under whatever notion of equality we happen to have) as

$$\texttt{let}\ b = N\ \texttt{in}\ \texttt{let}\ a = M\ \texttt{in}\ (a, b)$$

For instance, if our notion of effect is given by (not necessarily first-class) continuations and at least two different values that can be thrown, then terms M that throw cannot be central. We only need to take for N a term that throws something else in order to tell the difference between the two composites:

However, with first-class continuations, one can do much more than subject

M to testing for effects; one can actually reify M. For $N = \texttt{force}\ h$, the composite with `force` coming after M

$$\texttt{let}\ a = M\ \texttt{in}\ \texttt{let}\ b = \texttt{force}\ h\ \texttt{in}\ (a, b)$$

passes to h the continuation *after* running M. This gives access to the value that M returns after being run and possibly side-effecting. The composite with `force` coming first, by contrast, passes to h the continuation *before* M is computed. This has the same effect as wrapping the whole computation, include possible side-effects, into a thunk. Consider the example ML session in figure 1.

So instead of the somewhat weak argument "if M had effects, we should be able to find a test N that can tell the difference", we know that `force` will reify anything that follows. Now, intuitively speaking, in order for the two composites to agree, (i.e. for M thunked and unthunked to be the same) M itself must already be as good as reified.

6 Categorical Semantics

Given the notions of λ-abstraction from section 3.3, a simply-typed λ-calculus can be interpreted in a $\otimes\neg$-category. For call-by-value, control operators are naturally part of such a semantics, as they relate directly to the fundamental operations on the $\neg$ type. Specifically, `callcc` is interpreted as postcomposition with the adjoint correspondent

$$[\neg A \to A] = \neg(\neg A \otimes \neg A) \longrightarrow A$$

of the diagonal map $\neg A \longrightarrow \neg A \otimes \neg A$.

Definition 6.1 *Given a $\otimes\neg$-category $\mathcal{K}$ together with an interpretation $\mathcal{V}[\![b]\!]$ of base types, we can give an interpretation $\mathcal{V}[\![-]\!]$ for λ-calculus with control as follows. Types and environments are interpreted as usual, except for the breaking down of arrow types.*

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

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

$$\mathcal{V}[\![x_1 : \tau_1, \ldots x_n : \tau_n \vdash x_j : \tau_j]\!] \stackrel{\text{def}}{=} \pi_j$$

$$\begin{aligned}
\mathcal{V}[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] &\stackrel{\text{def}}{=} \overline{\lambda}_{\mathcal{V}[\![\sigma]\!]}\, \mathcal{V}[\![x : \sigma, \Gamma \vdash M : \tau]\!] \\
\mathcal{V}[\![\Gamma \vdash \texttt{throw}\ M\ N : \sigma]\!] &\stackrel{\text{def}}{=} \langle \mathcal{V}[\![\Gamma \vdash N : \tau]\!], \mathcal{V}[\![\Gamma \vdash M : \neg\tau]\!] \rangle ; \mathcal{V}[\![\tau]\!] \otimes \neg\pi_1 ; \mathsf{apply} \\
\mathcal{V}[\![\Gamma \vdash \texttt{callcc}\ M : \tau]\!] &\stackrel{\text{def}}{=} \mathcal{V}[\![\Gamma \vdash M : \neg\tau \to \tau]\!] ; \neg\langle \mathsf{id}_{\neg\mathcal{V}[\![\tau]\!]}, \mathsf{id}_{\neg\mathcal{V}[\![\tau]\!]} \rangle ; \mathsf{force}_{\mathcal{V}[\![\tau]\!]}
\end{aligned}$$

$$\mathcal{V}[\![\Gamma \vdash MN : \tau]\!] \stackrel{\text{def}}{=} \langle \mathcal{V}[\![\Gamma \vdash N : \sigma]\!], \mathcal{V}[\![\Gamma \vdash M : \sigma \to \tau]\!] \rangle ; \mathsf{apply}$$

Perhaps the most canonical property of control operators is the naturality of `callcc`, in the sense of the following axiom from [5].

$$V\ (\texttt{callcc}\ M) = \texttt{callcc}(\lambda k.V\ (M(\lambda x.k\ (Vx))))$$

where V ranges over values, i.e. $V ::= x \mid \lambda x.M$. However, this relies on continuations being a special case of procedures, as in Scheme. With a typing for continuations like that in NJ-SML, instances of this axiom will be ill-typed.

The negation operation suggested by our categorical semantics, definable as $\texttt{negate} \stackrel{\text{def}}{=} \lambda f.\lambda h.\texttt{callcc}((\texttt{throw}\, h) \circ f \circ \texttt{callcc} \circ \texttt{throw})$, is useful for adapting this axiom as follows

$$V\ (\texttt{callcc}\ M) = \texttt{callcc}(V \circ M \circ (\texttt{negate}\, V))$$

Example 6.2 *Let*

$$V = \texttt{fn n => n + 1}$$

and

$$M = \texttt{fn k => throw k 1}$$

Then V `(callcc` M`)` *and* `callcc(`V `o` M `o (negate` V`))` *both evaluate to 2.*

This axiom is sound for our semantics.

Proposition 6.3

$$\mathcal{V}[\![\Gamma \vdash V\ (\texttt{callcc}\ M) : \tau]\!] = \mathcal{V}[\![\Gamma \vdash \texttt{callcc}(V \circ M \circ (\texttt{negate}\, V)) : \tau]\!]$$

Definition 6.4 *The uncurrying call-by-name semantics relies on the variant λ-abstraction $\underline{\lambda}$ and application* $\underline{\underline{\mathsf{apply}}}$*. Let an interpretation $[\![b]\!]$ of base types be given.*

$$\mathcal{N}[\![\tau_1 \to \ldots \to \tau_n \to b]\!] \stackrel{\text{def}}{=} \neg(\mathcal{N}[\![\tau_1]\!] \otimes \cdots \otimes \mathcal{N}[\![\tau_n]\!] \otimes \neg[\![b]\!])$$

$$\mathcal{N}[\![x_1 : \tau_1, \ldots x_n : \tau_n \vdash x_j : \tau]\!] \stackrel{\text{def}}{=} \pi_j$$

$$\mathcal{N}[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] \stackrel{\text{def}}{=} \underline{\lambda}_{\mathcal{N}[\![\sigma]\!]} \mathcal{N}[\![x : \sigma, \Gamma \vdash M : \tau]\!]$$

$$\mathcal{N}[\![\Gamma \vdash MN : \tau]\!] \stackrel{\text{def}}{=} \langle \mathcal{N}[\![\Gamma \vdash N : \sigma]\!], \mathcal{N}[\![\Gamma \vdash M : \sigma \to \tau]\!] \rangle ; \underline{\underline{\mathsf{apply}}}$$

Proposition 6.5 *For the special case of the term model, that is $\mathcal{K} = \mathcal{K}(\mathrm{CPS})$, the denotation of a judgement is the equivalence class of its CPS transform.*

$$\mathcal{V}[\![\Gamma \vdash M : \tau]\!] = [(\!|\Gamma|\!), k : \neg(\!|\tau|\!) \vdash (\!|M|\!)(k)]$$
$$\mathcal{N}[\![\Gamma \vdash M : \tau]\!] = [\mathcal{N}(\!|\Gamma|\!), k : \neg\mathcal{N}(\!|\tau|\!) \vdash \mathcal{N}(\!|M|\!)(k)]$$

Acknowledgements
I am grateful to Phil Wadler and David N. Turner for initial discussions on the CPS calculus.

References

[1] Andrew Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] 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.

[3] Andrzej Filinski. Declarative continuations: an investigation of duality in programming language semantics. In D. H. Pitt et al., editors, *Category Theory and Computer Science*, number 389 in Lecture Notes in Computer Science, pages 224–249. Springer-Verlag, 1989.

[4] Andrzej Filinski. *Controlling Effects.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[5] Martin Hofmann. Sound and complete axiomatizations of call-by-value control operators. *Math. Struct. in Comp. Science*, 1994.

[6] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, 1996.

[7] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[8] John Power. Premonoidal categories as categories with algebraic structure. (submitted).

[9] John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science.* to appear.

[10] Guy Steele. Rabbit: A compiler for Scheme. Technical Report AI TR 474, MIT, May 1978.

[11] Guy Steele and Gerald Sussman. Lambda: The ultimate imperative. Technical Report AI Memo 353, MIT, March 1976.