# What is an Abstract Machine?

*Richard B. Kieburtz, Borislav Agapiev*

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 91-011

August, 1991

# What is an Abstract Machine?

*Richard B. Kieburtz*
*Borislav Agapiev*

Oregon Graduate Institute
19600 N.W. von Neumann Dr.
Beaverton, Oregon 97006 USA

*Technical Report CSE-091-011*

August, 1991

## Abstract

The thesis of this paper is that categorical models provide an appropriate framework for the high-level specification of computer architectures. As an example of this approach, we specify a categorical abstract machine capable of normal-order reduction of lambda calculus expressions to weak head-normal form. The paper includes substantial theoretical development of the appropriate categories and monads, including an account of involution, analogous to negation in intuitionistic logic.

An abstract machine is defined as a composite monad, a technique that emphasizes modularity of structure. In order to make control explicit in the machine model, the monad structure models continuations. This supports a formal specification of stored-program control. The categorical model is shown to be cartesian-closed. Finally, an implementation of (weak) lambda-calculus reduction by the categorical abstract machine is proved coherent with the syntactic reduction rule ($\beta$) of the calculus.

# What is an Abstract Machine?

*Richard B. Kieburtz*
*Borislav Agapiev*
Oregon Graduate Institute
19600 N.W. von Neumann Dr.
Beaverton, Oregon 97006

## 1. Abstract machines

Implementations of programming languages are often based upon an evaluation model called an 'abstract machine' [6, 12, 13, 15, 24, 27]. An abstract machine is a semantic model for the programming language, subject to informally imposed constraints that ensure its realizability. The utility of abstract machine models has been that they embody well defined concepts of implementation. They characterize particular strategies for evaluation, independently of the architecture of an underlying target machine that may be the ultimate host for a compiled implementation.

Typically, abstract machines have been specified either by a set of transitions on a rather complex machine state (this is usually called a register-transfer specification), or by a set of term rewriting rules that provide an operational model for a combinatory logic [25]. Either of these forms of specification can have *ad hoc* aspects. It is hardly ever clear why a particular machine state-space was chosen over possible alternatives, or why a particular set of combinators was selected.

With the Categorical Abstract Machine [4] we were given the idea that a formal characterization of an abstract machine may be possible by expressing computation in an appropriate category. The **CAM** was 'derived' from the categorical combinators of Curien [5]. The model is a cartesian-closed category; its morphisms are characterized as combinators, including an indexed family of de Bruijn combinators that destructure finitely nested pairs. The morphisms of this category form the basis for the instructions of the abstract machine. Computations are expressed as compositions of these morphisms. The equational theory of the categorical model entails a congruence relation on sequences of machine instructions. A naive compiler can be obtained immediately by giving a denotational semantics for a source language, with denotations in the category of computations.

However, upon closer examination, the **CAM** seems to have some problems: (i) The morphism *App* is taken as a primitive instruction of the **CAM**, yet no physical

realization is known for it (i.e. we do not know how to build an electronic device that directly realizes function application). (ii) The control structure of the **CAM** specifies conditional jumps, yet the underlying categorical model contains no morphisms that model control discontinuities. (iii) The **CAM** is a call-by-value machine and does not evaluate all functional programs that have normal forms when interpreted in a term-rewriting semantics. (iv) Recursive definitions are represented in the **CAM** by forming a cyclic data structure, which requires an updatable store that is completely outside the formal model. Thus while the **CAM** points the way towards abstract machines based upon precise, mathematical models for programming languages, there are gaps remaining to be filled.

Nevertheless, we believe the right answer to the rhetorical question posed in the title of the paper is that an abstract machine is an operational model that can be represented in a category whose morphisms form a basis for the instructions of the machine. Equations satisfied by the morphisms of the category provide a formal specification of an abstract machine. In this paper we construct a detailed categorical framework for evaluating expressions of the lambda-calculus, and use it to define an abstract machine that is essentially similar to the **CAM**. Its architecture is typical of machines that engineers know how to build. This machine has explicit control; it relies upon addressable control store; it evaluates the untyped lambda calculus to (weak) head normal forms and it resembles in most details other abstract machines that have been proposed for lazy evaluation of functional programming languages.[1] For an alternative approach, see [9,10], in which an abstract machine is derived from operational semantics of a language and its architecture refined through a series of transformations.

The model of computation we use is based upon the notion of continuations, which gives it the ability to describe explicit control transfers at the level of the machine. This is the principal design decision that affects the course of derivation of the mathematical model. Control is modeled in a category whose objects are continuations and whose morphisms are continuation transformers. It is the dual of a cartesian-closed category.

Our use of categorical duality was inspired by the research of Andrzej Filinski [7]. Rather than interpreting the objects and morphisms in a category uniformly as sets and

[1]Our abstract machine is not actually a 'lazy' evaluator but a call-by-need evaluator. In order to make it a lazy evaluator it would be necessary to add an explicit mechanism, such as updatable data store, with which to secure sharing of the results of previous evaluations.

functions, he also interprets objects as types of continuations and morphisms as continuation transformations. Values and continuations are categorical duals of one another. In terms of structure, the duals of products are coproducts (sums). A category can be closed with respect to its coproducts as well as with respect to products. We call the closure of coproducts co-exponentiation. It is the categorical mechanism that best models control abstractions.

We give a somewhat different interpretation to co-exponentials than does Filinski and show how co-exponentiation provides an appropriate mechanism to model the control structure of conventional computer architectures. We complete the picture by showing how the value-oriented semantics of functional programming languages are reflected by categorical duality into sequential computations expressed as continuations. Co-exponentials are used to effect transfers of control. Function application is emulated by a specific construction of abstract machine morphisms that formalize the familiar call and return mechanism of a von Neumann computer architecture.

This paper applies Moggi's idea that computation can be represented in a category with a suitable monad [20, 23]. The idea is that the essence of a notion of computation (state transformation, continuations, non-determinacy, etc.) is captured with a monad, and the detailed description of the possible computations is described with auxiliary functions, typed in the monad. Wadler [26] has given some nice examples of this technique. We explore in some detail the monad of continuations.

Section 2 summarizes the basic techniques of category theory used in the paper, and contains some informative examples. Section 3 is a study of the intuitionistic involution functor, analogous to negation in intuitionistic logic, and which induces the monad of continuations. This is all new material. Section 4 discusses strong monads and shows how a tensorial strength enables certain monad compositions. A principal result of this section is to show that the monad of continuations specifies a cartesian-closed subcategory, embedded in a category with products, sums, and a weaker form of closure. Section 5 provides a formal description of the categorical abstract machine (**CAM**), including its control structure. The categorical model is a composition of monads, each introducing an additional aspect of detail into the machine. Nearly all of this section can be read independently of the technical development in Sections 3 and 4. Section 6 presents a compilation scheme for the lambda calculus and proves that the resulting implementation by the **CAM** is coherent with the $\beta$-reduction rule of the calculus. Section 7 presents conclusions and points to directions for further work.

## 2. Categorical models

Categories provide a nice framework for specifying machine architectures at an abstract level for several reasons:

- An abstract machine should be a model for a programming language. With a categorical model, one can prove that the model is coherent with the logical specification of the language and the proof does not have to rely upon point-wise reasoning.
- Machine state, buried in the structure of objects of a category, can be as abstract as desired. The only aspects of the structure of an abstract machine that need to be revealed are those entailed by equations specifying its morphisms. These morphisms are the externally visible architecture of the machine — its instructions.
- Computation over some specified domain (of values, say) can be characterized by a monad, as described by Moggi [20]. This form of characterization is quite general, not at all restricted to any preconceived notion of computational model. To each monad there corresponds an adjunction that relates a category of values to a category of computations.
- Categories emphasize the compositional aspects of specification. Not only can complex instructions be composed of simpler ones, but entire substructures of a complex machine can be defined separately and composed to define a complete machine [21].

We assume the reader is familiar with the basic concepts of category theory such as arrows, functors, natural transformations, adjunction, initial and terminal objects, universal constructions, limits and the basic methods of proof. This section will review the definitions of adjunction and of monads, principally to establish notational conventions. It is not intended as a comprehensive introduction to the concepts.

*Definition 2.1*: An **adjunction** from a category $\mathbf{C}$ to a category $\mathbf{C}'$ consists of

(a) a pair of functors $F: \mathbf{C} \rightarrow \mathbf{C}'$ and $U: \mathbf{C}' \rightarrow \mathbf{C}$ called the *left-* and *right-adjoints*, respectively;

(b) a pair of natural transformations $\eta: Id_{\mathbf{C}} \dot{\rightarrow} UF$ and $\epsilon: FU \dot{\rightarrow} Id_{\mathbf{C}'}$ called the *unit* and *co-unit*;

(c) constraining equations $\epsilon F \circ F\eta = id$ and $U\epsilon \circ \eta U = id$.

□

We shall make use of a pair of facts about adjoints: a right adjoint functor carries all limits from $\mathbf{C}'$ to limits in $\mathbf{C}$ while a left adjoint functor carries colimits in the opposite direction ([17], pp. 114-115).

There are several alternative, but equivalent ways to define adjunction. One that is often useful is that of a (natural) bijection between the hom-sets of two categories,

$$\phi: \frac{\mathbf{C}'(FX \rightarrow Y)}{\mathbf{C}(X \rightarrow UY)}$$

Sometimes the names of the categories are omitted from this notation. This presentation emphasizes the role of the adjunction as relating the morphisms of two categories, and further allows us to deal with sets, as opposed to the objects of a category which are not always to be interpreted as sets.

**Notation**: The literature of category theory is rich in overloaded notation, which is confusing if not clearly understood. We shall use roman letters I, T, F, U (sometimes subscripted) to designate functors, and letters X, Y, Z to range over objects. Application of a functor to an object is designated by juxtaposition of symbols. So is functor composition. Thus the coding of symbols mentioned above is critical.

Lower case italic identifiers are used for functions (or morphisms). Ordinary composition of functions (morphisms) is designated by '$\circ$'. In sections 4 and 5, ';' is also used for composition in diagrammatic order in a dual category. Application of a functor to a morphism is indicated with parentheses around the argument.

Greek letters are used to name natural transformations and hom-set bijections. A hom-set bijection carries a morphism between adjunct categories. Applications of some particular hom-set bijections will be designated by an overbar or a superscripted or subscripted sharp symbol on the morphism identifier. Some authors omit these notational decorations but we find the potential for confusion to be too high.

A superscripted asterisk is used only to designate the Kleisli star extension of a function.

### 2.1. An example: cartesian closure

It is well known that cartesian-closed categories furnish categorical models for languages in which higher-order functions can be defined.

*Definition 2.2*: A category **C** is **cartesian-closed** if

i) There is a distinguished object, **1**,
for every object, $X \in \mathbf{C}$, a morphism $\Diamond_X : X \rightarrow \mathbf{1}$ exists,
for every morphism $h : X \rightarrow \mathbf{1}$, $h = \Diamond_X$.

ii) If X,Y are objects of **C**, then $X \times Y$ is an object,
there are morphisms $\pi_1 : X \times Y \rightarrow X$, $\pi_2 : X \times Y \rightarrow Y$ and a bifunctor
$\langle \_ , \_ \rangle : (Z \rightarrow X) \times (Z \rightarrow Y) \rightarrow (Z \rightarrow (X \times Y))$, satisfying

$$\pi_1 \circ \langle f, g \rangle = f$$
$$\pi_2 \circ \langle f, g \rangle = g$$
$$\langle \pi_1, \pi_2 \rangle = id_{X \times Y}$$

iii) If Y,Z are objects of **C**, then $[Y \Rightarrow Z]$ is an object.
For each object Y, there is a family of universal arrows $\{ap_{Y,Z} : ([Y \Rightarrow Z] \times Y) \rightarrow Z\}$
and for each arrow $f : (X \times Y) \rightarrow Z$ an arrow $f^{\#} : X \rightarrow [Y \Rightarrow Z]$ such that

$$ap_{Y,Z} \circ (f^{\#} \times id_Z) = f$$
$$(ap_{Y,Z} \circ (g \times id_Y))^{\#} = g$$

□

Cartesian closure is the existence of an adjunction from **C** to itself for each object Y, with left adjoint $F = (\_ \times Y)$ and right adjoint $U = (Y \Rightarrow \_)$. The natural bijection of hom-sets (in **C**) is

$$(\_)^{\#} : \frac{X \times Y \rightarrow Z}{X \rightarrow [Y \Rightarrow Z]}$$

where $[Y \Rightarrow Z]$ designates an object that may be interpreted as a function space.

The unit and co-unit, when specialized to an object, can be typed as $\eta_X : X \rightarrow Y \Rightarrow (X \times Y)$ and $\epsilon_Z : ([Y \Rightarrow Z] \times Y) \rightarrow Z$. The unit is the pair constructor, $\eta = \lambda x. \lambda y. \langle x, y \rangle$ and the co-unit is the family of universal arrows, $\{ap_{Y,Z}\}$, that realize function application. The adjunction 'internalizes' arrows of the category **C** as objects of the same category by enabling the elements of objects $[Y \Rightarrow Z]$ to be used as functions.

## 2.2. Monads

*Definition 2.3*: A **monad** over a category **C** is a triple, $\langle T, \eta, \mu \rangle$, where T is a functor from **C** to itself (an endofunctor) and $\eta : Id \dot{\rightarrow} T$, $\mu : T^2 \dot{\rightarrow} T$ are natural transformations satisfying the equations:

$$\mu_X \circ \mu_{TX} = \mu_X \circ T(\mu_X) \quad (1)$$
$$\mu_X \circ T(\eta_X) = id_{TX} \quad (2)$$
$$\mu_X \circ \eta_{TX} = id_{TX} \quad (3)$$

$\eta$ is called the **unit** and $\mu$ the **multiplier** of the monad.

□

Every monad $(T,\eta,\mu)$ induces an adjunction between suitably defined categories. One way to form an adjunction is to construct a Kleisli category, $\mathbf{C}_T$ from a category $\mathbf{C}$ with endofunctor[2] T. The left adjoint of a Kleisli adjunction is simply an inclusion functor, $I: \mathbf{C} \rightarrow \mathbf{C}_T$. The right adjoint is $T: \mathbf{C}_T \rightarrow \mathbf{C}$. Thus the adjunction defines the natural bijection of hom-sets

$$\frac{\mathbf{C}_T(IX \rightarrow Y)}{\mathbf{C}(X \rightarrow TY)}$$

We shall use an overbar to designate this bijection, so that if $f: X \rightarrow Y$ in $\mathbf{C}_T$, $\overline{f}: X \rightarrow TY$ in $\mathbf{C}$. Because of the inclusion functor, the objects of $\mathbf{C}_T$ are seen to be the the same as those of $\mathbf{C}$; its arrows correspond to the arrows of $\mathbf{C}$ that exist by virtue of the monad.

The image of an identity in $\mathbf{C}_T$ is a universal arrow, $\overline{id}_X = \eta_X: X \rightarrow TX$ in $\mathbf{C}$. A composition of arrows, $g \circ f$, where $f: X \rightarrow Y$ and $g: Y \rightarrow Z$ in $\mathbf{C}_T$ corresponds to a so-called Kleisli composition, $\mu_Z \circ T(\overline{g}) \circ \overline{f}: X \rightarrow TZ$ in $\mathbf{C}$.

*Definition 2.4*: The **Kleisli triple extension** of an arrow $g$ is defined to be $g^* = \mu_Z \circ T(\overline{g}): TY \rightarrow TZ$. From the monad axioms and the naturality of $\mu$ and $\eta$, it is straightforward to prove

$$f^* \circ \eta = \overline{f} \quad (1^*)$$
$$\eta^* \circ \overline{f} = \overline{f} \quad (2^*)$$
$$g^* \circ f^* = (g^* \circ \overline{f})^* \quad (3^*)$$

This gives a system of Kleisli triples, $(T, \eta, (\_)^*))$, where T is restricted to a function on objects. It provides an equivalent characterization of a monad.

□

The arrows of the Kleisli triple extension are just the arrows of the Kleisli category, projected into $\mathbf{C}$ by the functor T.

---

[2] Technically, the monad endofunctor is TI, which is abbreviated to T.

### The monad of cartesian closure

It is instructive to examine the Kleisli triples that correspond to the adjunction defining cartesian closure. Holding Y fixed, the object function is $T_{CC}X = Y \Rightarrow (X \times Y)$; the unit and multiplier are

$$\eta = \lambda x.\, \lambda y.\, x,y$$
$$\mu_X = \lambda t : T^2_{CC}X.\, \lambda y : Y.\, ap_{Y,X\times Y}(t\ y)$$

where $ap_{Y,X\times Y} : T_{CC}X \rightarrow X$ is the $X \times Y$-component of the co-unit of the adjunction.

If $f : X \rightarrow Z$ in $\mathbf{C}_T$, then

$$\eta_Z \circ f = \lambda x.\, \lambda y.\, fx, y : X \rightarrow T_{CC}Z$$

The image of $f$ in $\mathbf{C}$ is

$$T_{CC}(f) = \lambda t : T_{CC}X.\, \textbf{let}\ x, y' = t\ y\ \textbf{in}\ f\ x, y'$$

Composing the multiplier with $T_{CC}(\bar{f})$ gives

$$f^* : T_{CC}X \rightarrow T_{CC}Z = \mu_Z \circ T_{CC}(\bar{f}) = \lambda t : T_{CC}X.\, \lambda y : Y.\, \textbf{let}\ x, y' = t\ y\ \textbf{in}\ ap_{Y,Z\times Y}(\bar{f}\ x, y')$$

This suggests an operational interpretation of function application. Suppose a pair $(x,y)$ represents an environment structure in which $x$ designates the 'current' value and $y$ the remainder of the environment. An environment binds values to free variables, with correspondence established by position in a data structure of nested pairs. The unit of the monad injects a value into an environment by pairing. A function defined relative to an environment, expressed as $y : Y \vdash f : X \rightarrow Z$ in $\mathbf{C}_T$, corresponds in $\mathbf{C}$ to an explicitly curried function, $\bar{f}$, that when applied first to an argument, $x$, then to an environment, $y$, yields a new environment gotten by extending $y$ with $\bar{f}\ x\ y$.

### The monad of continuations

A further example is the monad of continuations,

$$TX = (X \Rightarrow A) \Rightarrow A$$
$$\eta = \lambda x.\, \lambda c.\, c\ x$$
$$\mu = \lambda t : T^2X.\, \lambda c : X \Rightarrow A.\, t\ (\lambda h : (X \Rightarrow A) \Rightarrow A.\, h\ c)$$

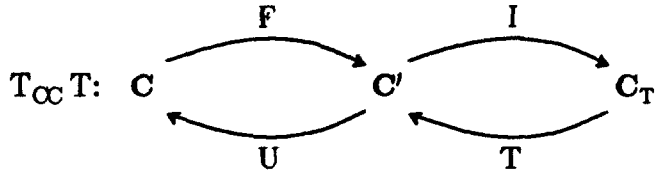
Let $f : X \rightarrow Y$ in $\mathbf{C}_T$ and $\bar{f} : X \rightarrow TY$ in $\mathbf{C}$. Then

$$T(f) = \lambda h : TX.\, \lambda c : Y \Rightarrow A.\, h\ (\lambda x.\, c\ (f\ x))$$
$$f^* = \mu_Z \circ T(\bar{f}) = \lambda h.\, \lambda c.\, h\ (\lambda x.\, \bar{f}\ x\ c)$$

Intuitively, $\eta_X$ specifies how a value of type X is included as a computation, while $(\_)^*$ specifies the extension of a function $\bar{f}$ from values to computations to a function $f^*$ from computations to computations.

### Call-by-value computation

A composite monad can be derived from the two previous examples by composing adjunctions.




The composite endofunctor on $\mathbf{C}$ is $T_V = UTIF$. The object function, unit and multiplier are

$$T_V X = Y \Rightarrow ((X \times Y) \Rightarrow A) \Rightarrow A$$

$$T_V(f) = \lambda t{:}T_V X.\, \lambda y{:}Y.\, \lambda c{:}(Z \times Y) \Rightarrow A.\, t\ y\ (\lambda h{:}X \times Y.\, c\ (f\ (\pi_1 h), \pi_2 h))$$
where $f : X \rightarrow Z$ in $\mathbf{C}_T$

$$\eta = \lambda x.\, \lambda y.\, \lambda c.\, c\ (x, y)$$

$$\mu = \lambda t{:}T_V^2 X.\, \lambda y{:}Y.\, \lambda c{:}(X \times Y) \Rightarrow A.\, t\ y\ (\lambda h{:}T_V X \times Y.\, \pi_1 h\ (\pi_2 h)\ c)$$

Proof the that the monad equations (1—3) are satisfied is obtained by calculation, substituting the definitions of the unit and multiplier into the equations and reducing the lambda-terms to normal forms.

Let $f : X \rightarrow Z$ in $\mathbf{C}_T$. Then we derive

$$f^* = \mu \circ T_V(\overline{f}) = \lambda t{:}T_V X.\lambda y{:}Y.\lambda c{:}(Z \times Y) \Rightarrow A.\, t\ y\ (\lambda h{:}X \times Y.\, ap\ ((\overline{f} \times id_Y)\ h)\ c)$$

In studying the final expression of $f^*$, we see that the argument $t$ is applied to an environment variable, $y$, yielding a computation of type $((X \times Y) \Rightarrow A) \Rightarrow A$. This computation is applied to a continuation gotten by abstracting $h : X \times Y$ from an expression representing the result of a computation. The argument variable, $h$, represents a structure that consists of nested pairs of values, not of computations. The computation described by this monad is call-by-value evaluation of applicative expressions.

### 2.2.1. T-cones and their limits

Certain covariant endofunctors construct limiting objects in **C**. Recall the definition of a cone [2]. Let $G$ be a graph, **C** be a category, and $\mathrm{D}: G \rightarrow \mathbf{C}$ be a diagram in **C** with shape $G$. A **cone** with base D is a pair $\langle \mathrm{X}, \{p_a\} \rangle$, where $\mathrm{X} \in Obj(\mathbf{C})$ is the **vertex** and $\{p_a\}$ is a family of arrows in **C** indexed by nodes of $G$, such that $p_a : \mathrm{X} \rightarrow \mathrm{D}a$ for each node $a$ of $G$. A cone is said to be **commutative** if the diagram $\mathrm{D}(G)$ commutes in **C**. (In case the diagram is discrete, the commutative property holds trivially.) A cone generalizes the notion of a family of projections. A commutative cone with base D and vertex X is called a **limit** of the diagram D if from every cone $c$ over the same base D, there is a unique arrow from $vertex(c)$ to X. The vertex of a limiting cone is called the limit object, or simply the limit if it is understood that the subject of discussion is an object.

**Example 1**: The prototypical example of a limit is a binary product. Take $G$ to be the two-point, discrete graph, D to be a map to two objects X and Y of **C**, and the limiting cone with base D is $\langle \mathrm{X} \times \mathrm{Y}, \{\pi_1, \pi_2\} \rangle$.

**Example 2**: More generally, consider limits with parameters [17]. Let the diagram D be parametric on an object of **C**; $\mathrm{D}: \mathbf{C} \rightarrow G \rightarrow \mathbf{C}$. With the two-point graph of the previous example, introducing a parameter allows us to fix one of the objects in the codomain of D, so that D(A) is a map to objects A and X, and the limiting cone with base D(A) is $\langle \mathrm{A} \times \mathrm{X}, \{\pi_1, \pi_2\} \rangle$.

**Example 3**: A more interesting example occurs when a graph $G$ is also obtained from the object parameter of D. For example, in a cartesian-closed category whose objects are sets, the graph of an object is just the discrete graph of its elements. If we take the diagram $\mathrm{D}(\mathrm{A}) = G(\mathrm{A}) \rightarrow \mathrm{X}$, then the limiting cone with base D(A) is $\langle [\mathrm{A} \Rightarrow \mathrm{X}], \{ap_{\mathrm{X}}(\_, a) \mid a \in \mathrm{A}\} \rangle$

**Example 4**: In a cartesian-closed category let $\mathrm{D}(\mathrm{A}) = \mathbf{C}(\mathrm{X}, \mathrm{A}) \rightarrow \mathrm{A}$ (by an abuse of notation identifying the hom-set with its graph). The limiting cone with base D(A) is $\langle (\mathrm{X} \Rightarrow \mathrm{A}) \Rightarrow \mathrm{A}, \{ap_{\mathrm{A}}(\_, c) \mid c \in [\mathrm{X} \Rightarrow \mathrm{A}]\} \rangle$.

When the objects TX of a category with a monad are limit objects, we say that T has limits and shall refer to the limit cones as T-cones, rather than specifying the base of each cone. When T has limits, there is an arrow $\mathrm{Z} \rightarrow \mathrm{TX}$ which is the mediating morphism of a T-cone extending from the vertex Z to a base that it shares with the limiting T-cone.

### 2.2.2. The structure of T(C)

The codomain of an endofunctor T is a subcategory of **C**, which we designate as T(**C**). When T is an endofunctor component of a monad, T(**C**) can have interesting properties.

A Kleisli adjunction $\langle F_T, U_T, \eta, \epsilon \rangle$: $\mathbf{C} \rightarrow \mathbf{C}_T$ is derived from a monad on **C**, but it also induces a comonad $\langle F_T U_T, \epsilon, \nu \rangle$ on $\mathbf{C}_T$, where $\nu = F_T \eta U_T$: $F_T U_T \dot{\rightarrow} F_T U_T F_T U_T$. (It is straightforward to check that the comonad laws are satisfied.)

When T maps objects to limit objects in **C**, the comonad on $\mathbf{C}_T$ reveals a great deal about the structure of T(**C**). A T-coalgebra is associated with a comonad, dually as a T-algebra is associated with a monad ([17], Sec. VI.2).

*Definition 2.5*: If $T = \langle T, \epsilon, \nu \rangle$ is a comonad on $\mathbf{C}_T$, a T-coalgebra $\langle X, k_X \rangle$ is represented as a pair where $X \in Obj(\mathbf{C}_T)$ is the carrier of the algebra and the arrow $k_X$: $X \rightarrow TX$ in **C** (called the co-structure map) obeys the equations:[3]

$$\epsilon_{F_T X} \circ F_T k_X = id_{F_T X}$$
$$F_T(T(k_X) \circ k_X) = \nu_{F_T X} \circ F_T k_X$$

A T-algebra homomorphism is an arrow $f : X \rightarrow Y$ of **C** such that

$$k_Y \circ f = T(f) \circ k_X$$

□

*Proposition 2.6*: To every $X \in Obj(T(\mathbf{C}))$ there corresponds a T-coalgebra $\langle X, \eta_X \rangle$.

*Proof*: It is easily verified that $\eta_X$ is a T-co-structure map, using the first triangle law of an adjunction and the formula for $\nu$ from the definition of a comonad.

□

*Proposition 2.7*: Every arrow of **C** is a T-coalgebra homomorphism.

*Proof*: If $f : X \rightarrow Y$ in **C** then

$$T(f) \circ \eta_Y = \eta_X \circ f \qquad (\eta \text{ is natural})$$

□

---

[3] The functor $F_T$ is often omitted from these equations as it is simply an injection functor. Its effect on a T-coalgebra is to drop the construction map, $F_T\langle X, k_X \rangle = X$.

The objects of T(**C**) are isomorphic to the T-coalgebras of **C**. The arrows, $X \rightarrow Y$, of the Kleisli category $\mathbf{C}_T$ are isomorphic (by the natural isomorphism $\phi_T$ of the adjunction) to mediating arrows, $X \rightarrow TY$, of T-cones in **C**. When T has limits, the components of the unit of the Kleisli adjunction, $\eta_X : X \rightarrow TX$, which are the isomorphic images of the identity arrows of $\mathbf{C}_T$, are unique limit constructors.

*Proposition 2.8*: When an endofunctor T has limits, each object in the codomain of $U_T$ is a limit object of a family of T-cones.

*Proof*: Every arrow of $\mathbf{C}_T$ is isomorphic (by $\phi_T$) to the mediating arrow of a T-cone of **C**. Since $U_T$ is a right adjoint functor, it preserves all limits of $\mathbf{C}_T$, and in particular, preserves the limits of its families of T-cones.

□

*Theorem 2.9*: If $T : \mathbf{C} \rightarrow \mathbf{C}$ has limits then T(**C**) is a full subcategory of **C**.

*Proof*: T(**C**) is a category as it is defined to be the codomain of the functor $U_T$ and its objects are the set $\{TX \mid X \in Obj(\mathbf{C})\}$ by Proposition 2.8. Let $g : TX \rightarrow TY$ in **C**. Then $\phi_T^{-1}(g) : TX \rightarrow Y$ in $\mathbf{C}_T$ and $g^* = U_T(\phi_T^{-1}(g)) : T^2X \rightarrow TY$ in **C**. By (K2), $g = g^* \circ \eta_{TX} = U_T(\phi_T^{-1}(g)) \circ \eta_{TX}$. Thus $g$ belongs to T(**C**).

□

## 3. Involution

*Definition 3.1*: An **involution** of a category **C** is a pair, (D,A), where $D : \mathbf{C} \rightarrow \mathbf{C}^{op}$ is a faithful, contravariant functor and $A \in Obj(\mathbf{C})$ has the property that DA is terminal in **C**.

□

Recall that a functor is faithful if it does not identify parallel arrows that are distinct in **C**. The composition of faithful functors is faithful, thus an involution functor from **C** to $\mathbf{C}^{op}$ composed with the reverse involution from $\mathbf{C}^{op}$ to **C** defines a self-embedding of **C**. The functor whose object mapping is the identity and which simply reverses the sense of every arrow is an example of an involution functor, albeit a trivial one.

The role of the involution object will be illustrated by a particular involution that we shall investigate. However, note that the involution object has the property that

$$\forall\, X \in Obj(\mathbf{C}).\ |\mathbf{C}(A,X)| \leq 1$$

as an immediate consequence of the definition.

We are interested in a particular involution, that which generates continuations as the objects dual to value objects. Toward this end, we shall explore a closure property of certain cartesian categories, weaker than cartesian closure.

*Definition 3.2*: A category **C** has a **quasi-closure** if it has direct products and a distinguished object A such that the following bijection, natural in X and Y, holds:

$$\psi: \frac{\mathbf{C}(X \times Y, A)}{\mathbf{C}(X, Y \Rightarrow A)}$$

□

*Theorem 3.3*: Let **C** be a category with a quasi-closure. Then there is a family of universal arrows, $\{ans_Y: (Y \Rightarrow A) \times Y \rightarrow A\}$ such that every morphism $f: X \times Y \rightarrow A$ can be factored through $ans_Y$.

*Proof*: Consider the bijection of composites,

$$\frac{X \times Y \xrightarrow{g \times id_Y} Z \times Y \xrightarrow{\psi^{-1}(h)} A}{X \xrightarrow{g} Z \xrightarrow{h} Y \Rightarrow A}$$

$$\psi(\psi^{-1}(h) \circ (g \circ id_Y)) = h \circ g$$

Let $Z = Y \Rightarrow A$ and $h = id_{Y \Rightarrow A}$.
Then $g = \psi(\psi^{-1}(id_{Y \Rightarrow A}) \circ (g \times id_Y))$ and $ans_Y = \psi^{-1}(id_{Y \Rightarrow A})$.
Let $g = \psi^{-1}(f): X \rightarrow (Y \Rightarrow A)$, where $f: X \times Y \rightarrow A$ and $f = ans_Y \circ (g \times id_Y)$.
□

*Corollary 3.4*: For a quasi-closed category, **C**, the following assertions are equivalent:

a) $ans_A$ is an isomorphism;

b) $A \Rightarrow A$ is a terminal object;

c) $\forall\ X \in Obj(\mathbf{C})$. there is a unique arrow $\delta_X: X \times A \rightarrow A$.

*Proof*: (b) $\Rightarrow$ (a): $\Diamond_X: X \rightarrow [A \Rightarrow A]$ is a unique arrow, by hypothesis (b). Since $ans_A$ is universal,

$$ans_A \circ (\Diamond_A \times id_A) = \psi_A^{-1}(\Diamond_A)$$

But $\psi_A^{-1}(\Diamond_A)$ is the unique morphism of type $A \times A \rightarrow A$.
Thus $ans_A \circ (\Diamond_A \times id_A) = (\pi_2)_{A,A} = \pi_2 \circ (\Diamond_A \times id_A)$ from which we conclude that
$ans_A \circ \langle \Diamond_A, id_A \rangle = id_A$ and $ans_A = (\pi_2)_{A \Rightarrow A, A}$.

Composing with $ans_A$ on the right,

$$\begin{aligned}\langle \Diamond_A, id_A \rangle \circ ans_A &= \langle \Diamond_A \circ ans_A, ans_A \rangle \\ &= \langle \Diamond_{|A \Rightarrow A| \times A}, (\pi_2)_{A \Rightarrow A, A} \rangle \\ &= (id_{A \Rightarrow A} \times id_A) \\ &= id_{|A \Rightarrow A| \times A}\end{aligned}$$

(c) $\Rightarrow$ (b): For each $X \in Obj(\mathbf{C})$ $\psi_X(\delta_X): X \rightarrow [A \Rightarrow A]$ is unique, as $\delta_X$ is unique by hypothesis (c) and every arrow with codomain $[A \Rightarrow A]$ is introduced by exponentiation.

(a) $\Rightarrow$ (c): $ans_A$ is an isomorphism by hypothesis. Its inverse is easily demonstrated:

$$ans_A \circ \langle \psi(\pi_2)_{A,A}, id_A \rangle = ans_A \circ (\psi(\pi_2)_{A,A} \times id_A) \circ \langle id_A, id_A \rangle = (\pi_2)_{A,A} \circ \langle id_A, id_A \rangle = id_A$$

Thus $ans_A^{-1} = \langle \psi(\pi_2)_{A,A}, id_A \rangle$ and $ans_A = (\pi_2)_{A \Rightarrow A, A}$. This allows $\delta_X$ to be calculated. There must exist an arrow of type $X \times A \rightarrow A$ for $(\pi_2)_{X,A}$ is one. Suppose $\delta_X$ is any arrow of this type. Then

$$\begin{aligned}\delta_X &= ans_A \circ (\psi_X(\delta_X) \times id_A) \\ &= (\pi_2)_{A \Rightarrow A, A} \circ (\psi_X(\delta_X) \times id_A) \\ &= id_A \circ (\pi_2)_{X,A} \\ &= (\pi_2)_{X,A}\end{aligned}$$

□

*Definition 3.5*: We say that $A \in Obj(\mathbf{C})$ is an **involution object** if $\forall X \in Obj(\mathbf{C})$, two conditions hold:

i) there is a unique morphism $\delta_X: X \times A \rightarrow A$,

ii) there is a monic morphism $\kappa_X: X \rightarrow A$.

□

Clause (i) of the above definition is motivated by Corollary 3.4. Of the three equivalent conditions given there, 3.4(c) is the only one that does not depend upon properties of quasi-closure.

*Proposition 3.6*: In a quasi-closed category with an involution object A, there is a 1-1 correspondence between arrows $X \rightarrow A$ and the elements of $[X \Rightarrow A]$,

*Proof*: (Here we designate the terminal object $[A \Rightarrow A]$ by **1**.)

$$\dfrac{\dfrac{\dfrac{X \xrightarrow{k} A}{X \xrightarrow{k \circ E_X^{-1}} A}}{1 \times X \xrightarrow{k \circ E_X^{-1}} A}}{1 \xrightarrow{\psi(k \circ E_X^{-1})} X \Rightarrow A}$$

in which $E_X$ is the 'administrative' morphism of left identity in a cartesian category.

□

*Theorem 3.7* Let **C** be a quasi-closed category with an involution object A. Define a functor D by extending the object mapping

$$D = (\_ \Rightarrow A) : \mathbf{C} \rightarrow \mathbf{C}^{op}$$

to hom-sets in the usual way. Then (D,A) is an involution of **C**.

*Proof*: From Corollary 3.4, $DA = [A \Rightarrow A]$ is a terminal object. It is immediate from Definition 3.2 that an element of $[X \Rightarrow A]$ is contravariant in X. It remains to show that D is faithful.

Let $f, g : X \rightarrow Y$ be parallel arrows in **C** and $D(f), D(g) : DY \rightarrow DX$ be their images in **C**. If $D(f) = D(g)$ then $\forall\, h : Z \rightarrow DY.\ D(f) \circ h = D(g) \circ h$. In particular, choose $Z = \mathbf{1}$ and let $h = \psi(k \circ E_Y^{-1})$, where $k : Y \rightarrow A$. Then

$$D(f) = D(g) \Rightarrow \forall\, k : Y \rightarrow A.\ D(f) \circ \psi(k \circ E_Y^{-1}) = D(g) \circ \psi(k \circ E_Y^{-1})$$

Naturality of $\psi^{-1}$, expressed in the following diagram,

$$\begin{array}{ccc} \mathbf{C}(\mathbf{1}, DY) & \xrightarrow{\psi^{-1}} & \mathbf{C}(\mathbf{1} \times Y, A) \\ \Big\downarrow{\scriptstyle \mathbf{C}(\mathbf{1}, D(f))} & & \Big\downarrow{\scriptstyle \mathbf{C}(f, A)} \\ \mathbf{C}(\mathbf{1}, DX) & \xrightarrow{\psi^{-1}} & \mathbf{C}(\mathbf{1} \times X, A) \end{array}$$

gives

$$\psi^{-1}(\mathrm{D}(f)\circ\psi(k\circ E_{\mathrm{Y}}^{-1}))=k\circ E_{\mathrm{Y}}^{-1}\circ(\mathbf{1}\times f)=k\circ f\circ E_{\mathrm{X}}^{-1}$$

Thus $\mathrm{D}(f)=\mathrm{D}(g)\Rightarrow\forall\ k:\mathrm{Y}\rightarrow\mathrm{A}.\,k\circ f=k\circ g$.

Furthermore, from Definition 3.2 and the definition of D we have that

$$\forall\ k:\mathrm{X}\rightarrow\mathrm{A}.\,\mathrm{D}(k)=\psi(k\circ E_{\mathrm{X}}^{-1})$$

which induces the hom-set isomorphism

$$\mathbf{C}(\mathrm{X},\mathrm{A})\cong\mathbf{C}(\mathbf{1},\mathrm{DX})\cong\mathrm{DX}.$$

This justifies the final step needed to establish that D is faithful, namely that

$$(\forall\ k:\mathrm{Y}\rightarrow\mathrm{A}.\,k\circ f)=k\circ g\Rightarrow f=g$$

for since A is an involution object, there is a monic arrow of type Y→A.

□

Here we consider some consequences of this involution.

*Corollary 3.8*: The involution functor D from a quasi-closed category **C** to $\mathbf{C}^{\mathrm{op}}$ is self-adjoint.

*Proof*: Consider the sequence of natural bijections in **C**,

$$\frac{\dfrac{\dfrac{\dfrac{\mathrm{X}\xrightarrow{\psi(h)}\mathrm{DY}}{\mathrm{X}\times\mathrm{Y}\xrightarrow{h}\mathrm{A}}}{\mathrm{Y}\times\mathrm{X}\xrightarrow{h\circ C}\mathrm{A}}}{\mathrm{Y}\xrightarrow{\psi(h\circ C)}\mathrm{DX}}$$

where $C_{\mathrm{X,Y}}:\mathrm{X}\times\mathrm{Y}\rightarrow\mathrm{Y}\times\mathrm{X}$ is the exchange morphism for symmetric products. We shall call the composite bijection $\phi$, and write it as a bijection between hom-sets of $\mathbf{C}^{\mathrm{op}}$ and **C**,

$$\phi_{\mathrm{X,Y}}:\frac{\mathbf{C}^{\mathrm{op}}(\mathrm{DX}\rightarrow\mathrm{Y})}{\mathbf{C}(\mathrm{X}\rightarrow\mathrm{DY})}$$

This formulation makes it apparent that $\phi$ expresses the self-adjunction of D.

□

The bijection, when represented entirely in **C**, can be expressed in lambda-notation,

$$\phi(f)=\lambda y.\,\lambda x.f\ x\ y$$

from which we see that $\phi^{-1}=\phi$ as untyped expressions.

There is an analogy between involution in a cartesian-closed category and negation in intuitionistic logic. In intuitionistic logic, absurdity is expressed by the closed, second-order formula $\forall$A.A, where A ranges over all propositions. To assert the negation of a proposition X, we write the implication X$\Rightarrow\forall$A.A. It is customary to replace $\forall$A.A by a special symbol $\perp$ when second-order quantification is not used elsewhere, obtaining first-order intuitionistic logic with (weak) negation. The implication $\perp\Rightarrow$X is the X-component of an axiom scheme; it has a unique (i.e. trivial) proof object. We shall not press the analogy further here, other than to note that if a category has only quasi-closure and not cartesian-closure, then it does not model the rule of modus ponens in logic.

### 3.1. Naturality properties

The properties of a natural bijection between categories are summarized in four equations in [17]. Specialized to the adjunction (D,D), and using the symbol ';' for composition in $\mathbf{C}^{op}$, the equations are:

$$\phi(f\,;\mathrm{D}(h)) = \phi(f)\circ h \qquad \phi(k\,;f) = \mathrm{D}(k)\circ\phi(f) \tag{4a,4b}$$

$$\phi^{-1}(g\circ h) = \phi^{-1}(g)\,;\mathrm{D}(h) \qquad \phi^{-1}(\mathrm{D}(k)\circ g) = k\,;\phi^{-1}(g) \tag{5a,5b}$$

Choosing $f = id$ in (4b) gives

$$\phi(k) = \mathrm{D}(k)\circ\eta \tag{6}$$

where $\eta$ is the unit of the adjunction. Choosing $k = \phi^{-1}(f)$ in (5b) gives

$$\phi^{-1}(\mathrm{D}(\phi^{-1}(f))\circ g) = \phi^{-1}(f)\,;\phi^{-1}(g) \tag{7}$$

The endofunctor $\mathrm{T} = \mathrm{D}^2$ induces the monad of continuations, seen previously as an example, on **C**. The adjunction between **C** and its Kleisli category is not the same as the self-adjunction of the functor D. Nevertheless, we shall see that the categories $\mathbf{C}_\mathrm{T}$ and $\mathbf{C}^{op}$ are related.

The bijection $\phi$ allows the identities and compositions of $\mathbf{C}_\mathrm{T}$ to be expressed in $\mathbf{C}^{op}$. Letting $\overline{q} = \phi_{X,Z}(k)$ in (6), we have that

$$\overline{f} = \mathrm{D}(\phi^{-1}_{X,Z}(\overline{f}))\circ\eta_X$$

But $\overline{f} = f^*\circ\eta_X$ is a law of any monad, and $\eta_X$ is a universal arrow, so

$$f^* = \mathrm{D}(\phi^{-1}_{X,Z}(f)). \tag{8}$$

*Proposition 3.9*:

$$(i) \quad \phi^{-1}(\eta) = id$$
$$(ii) \quad \phi^{-1}(f^* \circ g) = \phi^{-1}(f); \phi^{-1}(g)$$

*Proof*:

$(i)$ $\eta$ is the unit of the adjunction (D,D), whose natural bijection is $\phi$.

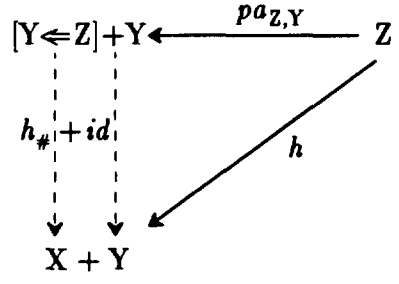
$(ii)$ Immediate from (7) and (8)

□

The preceding proposition establishes an embedding of the Kleisli category, $\mathbf{C}_T$, which was introduced to represent functions as computations, into the category $\mathbf{C}^{op}$, in which the arrows of $\mathbf{C}_T$ are represented as *continuation transformers*, embedded by involution. We shall show how function application can be represented with continuation transformers.

### 3.2. Co-exponentiation in the Kleisli category of continuations

Coproduct closure is an isomorphism dual to cartesian (direct) product closure and is illustrated by the commutation of the following diagram:




Here, $pa_{Z,Y}$ is a universal arrow called *co-application* and $[Y \Leftarrow Z]$ designates a co-

exponential object, a type whose elements are co-abstractions. The arrow $h_{\#}$ is the co-curried image of $h$ under the co-closure.

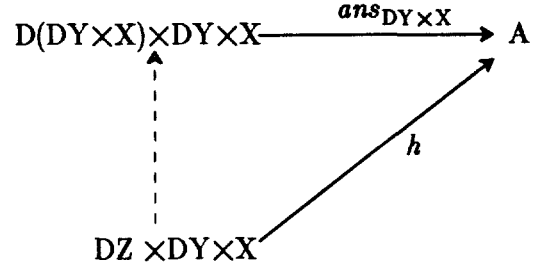
Technically, we require the isomorphism (natural in X and Y) between D(X+Y) and DX×DY to be made explicit. It is

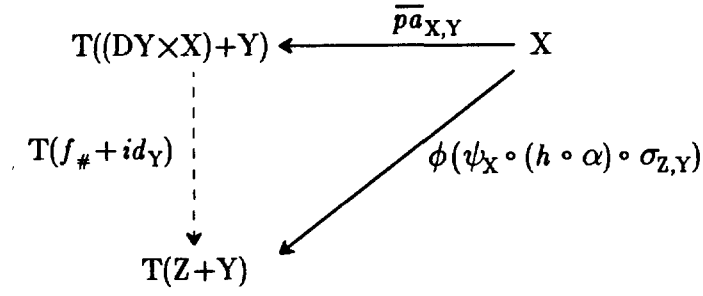
$$\sigma_{X,Y} = \lambda c : D(X+Y).(c \circ inl, c \circ inr)$$
$$\sigma^{-1}_{X,Y} = \lambda c' : DX \times DY. [\pi_1 c', \pi_2 c']$$

The following theorem restates a result of Moggi and Agapiev [22] for the case of quasi-closed categories.

*Theorem 3.10*: If **C** has finite products and coproducts and is quasi-closed, then $\mathbf{C}_T$ has co-exponentials.

*Proof*: The diagram expressing the universal property of the arrow $ans_{DY \times X}$ can be transformed isomorphically in **C**.




where $f = \phi(h \circ \sigma_{Z,Y})$

and $\quad \overline{pa}_{X,Y} = \phi(\psi_X(ans_{DY \times X} \circ \alpha) \circ \sigma_{DY \times X,Y})$.

and $\quad \alpha = \langle \langle \pi_1, \pi_1 \circ \pi_2 \rangle, \pi_2 \circ \pi_2 \rangle$

Universality of $\overline{pa}$ is assured by the universality of *ans*, which is preserved by the isomorphisms. The co-exponential object, viewed in $\mathbf{C}_T$, is $DY \times X$. When $f : X \rightarrow Z + Y$, the corresponding co-curried morphism is $f_{\#} : (DY \times X) \rightarrow Z$ in $\mathbf{C}_T$.

□

There is a computational interpretation of co-application. Just as the co-unit of the adjunction of cartesian closure, *ap*, provides the morphism that models substitution of a value for a bound variable in the body of an expression, co-application models substitution of continuations in an expression with a bound continuation variable. $\phi^{-1}(\overline{pa}_{X,Y}) \circ \sigma^{-1}_{DY \times X, Y} : D(DY \times X) \times DY \rightarrow DX$ in $\mathbf{C}^{op}$ can be seen as taking a pair, consisting of a continuation transformer and a continuation for the anticipated result of a function's application, into a new continuation, that which accepts the argument. This is, of course, an interpretation that corresponds to reading a program backwards from its end towards its beginning. When reading the same program forwards, $\overline{pa}_{X,Y}$ can be interpreted as taking an argument of type X into a *context* for a pair that consists of a continuation transformer and a result continuation. A context for a pair has a formal representation as a computation whose type is a sum, although this is less intuitive.

## 4. Strong monads

A monad $(T, \eta, \mu)$ on a category $\mathbf{C}$ with direct products is said to be *strong* if there is a family of morphisms $\{T_{X,Y} : TX \times TY \rightarrow T(X \times Y)\}$, natural in X and Y. $T$ is called a **symmetric tensorial strength** for the monad.

A cartesian category equipped with a strong monad of continuations can model call-by-value computation [23]. The counterpart of this strength in a model for call-by-need computation is simply the unit of the monad, specialized to products, $\eta_{TX \times TY} : TX \times TY \rightarrow T(TX \times TY)$. We shall exhibit the symmetric tensorial strength, even though it is not required to model call-by-need computation with non-strict pairing.

*Proposition 4.1*: Let $\mathbf{C}$ be a quasi-closed category with the monad of continuations. This monad has a symmetric tensorial strength, $T$. Furthermore, there is a family of morphisms, $\{\tilde{T} : T(X \times Y) \rightarrow TX \times TY\}$, natural in X and Y, which is a quasi-inverse of $T$.

*Proof*: We shall first derive $\tilde{T}$. An explicit representation is derived by passing a direct product object through the involution functor twice. The two steps of involution realize the DeMorgan laws of classical propositional logic.

First step: Let $\pi_1$ designate the first projection of the product in the Kleisli category, $\mathbf{C}_T$. Then $\overline{\pi}_1 : X \times Y \rightarrow TX$ in $\mathbf{C}$, and thus $\phi^{-1}(\overline{\pi}_1) : DX \rightarrow D(X \times Y)$ in $\mathbf{C}^{op}$. Taken together with the image of the second projection, this gives

$$[\phi^{-1}(\overline{\pi}_1), \phi^{-1}(\overline{\pi}_2)] : DX + DY \rightarrow D(X \times Y)$$

Second step: $\sigma_{X,Y} : D(X+Y) \rightarrow DX \times DY$

Now the morphisms derived in these two steps should be composed in $\mathbf{C}$. We take the involution of the morphism derived in the first step to get

$$\tilde{T} = \sigma_{DX,DY} \circ D([\phi^{-1}(\overline{\pi}_1), \phi^{-1}(\overline{\pi}_2)]) : T(X \times Y) \rightarrow TX \times TY$$

which can also be expressed in lambda notation as[4]

$$\tilde{T} = \lambda t{:}T(X \times Y).(\lambda c{:}DX.t\,(c \circ \pi_1), \lambda c{:}DY.t\,(c \circ \pi_2))$$

To carry the proof in the other direction requires a morphism called simply a **tensorial strength** [20].

$$t_{X,Y} : X \times TY \rightarrow T(X \times Y)$$

A tensorial strength for T exists in a quasi-closed category. Its representation as a lambda-term is:

$$t_{X,TY} = \lambda p{:}X \times TY.\lambda k{:}X \times Y \Rightarrow A.\pi_2\, p\,(\lambda y{:}Y.\, k\,(\pi_1\, p, y))$$

Two applications of the tensorial strength are required in order to reduce both components of a pair, yielding

$$T = (t_{X,Y} \circ c_{TY,X})^* \circ t_{TY,X} \circ c_{TX,TY} : TX \times TY \rightarrow T(X \times Y)$$

which in lambda notation is

$$T = \lambda p{:}TX \times TY.\lambda c{:}D(X \times Y).\, \pi_1\, p\,(\lambda x.\ \pi_2\, p\,(\lambda y.\ c\,(x,y)))$$

□

The composition of the symmetric tensorial strength with its quasi-inverse on the left is

---

[4] Technically, lambda calculus is only justified as the language in which to express morphisms of a cartesian-closed category. It can also be used with a category that is quasi-closed if applications are always denoted by value elements of the involution object, A.

$$\tilde{T} \circ T = \lambda p{:}\mathrm{TX}\times\mathrm{TY}.(\lambda c{:}\mathrm{DX}.\,\pi_1\,p\,(\lambda x{:}X.\,\pi_2\,p\,(\mathrm{K}\,(c\,x))),\lambda c{:}\mathrm{DY}.\,\pi_1\,p\,(\mathrm{K}\,(\pi_2\,p\,c))) \qquad (9)$$

where K is the weakening combinator, $\mathrm{K}\,x\,y = x$. $\tilde{T}_{\mathrm{X,Y}}$ is an approximate left inverse to $T_{\mathrm{X,Y}}$. It fails to invert $T_{\mathrm{X,Y}}$ when one of the computations TX or TY fails to converge. To show the inverse property, it is convenient to define a subcategory of **C** that may be interpreted as a category of total computations, although this subcategory is not one in which constructive models can be found.

*Definition 4.2*: Let **C** be a quasi-closed, cartesian category with the monad of continuations. Then $\mathbf{C}_{total}$ is the subcategory of **C** that is inductively defined by the following conditions:

1) $Obj(\mathbf{C}_{total}) = Obj(\mathbf{C})$

2) $id_{\mathrm{X}} \in \mathbf{C}_{total}(\mathrm{X,X})$ for all $\mathrm{X} \in Obj(\mathbf{C}_{total})$

3) $f \in \mathbf{C}_{total}(\mathrm{X,TY})$ if $f \in \mathbf{C}(\mathrm{X,TY})$ and $\phi^{-1}(f) \in \mathbf{C}(\mathrm{DY,DX})$ is monic

4) $f \in \mathbf{C}_{total}(\mathrm{X,Y})$ if $f \in \mathbf{C}(\mathrm{X,Y})$ and $\mathrm{D}(f) \in \mathbf{C}(\mathrm{DY,DX})$ is monic

□

The intuition that underlies the monic requirement is that monic morphisms, when interpreted as functions, do not discard their arguments. Thus a monic morphism of type $\mathrm{X} \rightarrow \mathrm{Y}$ can be interpreted as a strict function, or if its type is $\mathrm{DX} \rightarrow \mathrm{A}$ so that its argument is a continuation, it can be interpreted as a total computation. There are a number of immediate consequences of Definition 4.2.

*Proposition 4.3*:

A. $1 = \mathrm{A} \Rightarrow \mathrm{A}$ is a terminal object of $\mathbf{C}_{total}$.

B. $k \in \mathbf{C}_{total}(\mathrm{X,A})$ if $k \in \mathbf{C}(\mathrm{X,A})$

C. the elements of TX in $\mathbf{C}_{total}$ are isomorphic to the monic arrows $k_{\mathrm{X}} \in \mathbf{C}(\mathrm{DX,A})$

D. $\eta_{\mathrm{X}} \in \mathbf{C}_{total}(\mathrm{X,TX})$

E. there is a family of morphisms $\{eval_{\mathrm{X}} \in \mathbf{C}_{total}(\mathrm{TX,X})\}$, such that $eval_{\mathrm{X}}$ is inverse to $\eta_{\mathrm{X}}$, for all $\mathrm{X} \in Obj(\mathbf{C}_{total})$

F. for $f \in \mathbf{C}_{total}(\mathrm{X,TY})$, condition 4.2(4) is equivalent to 4.2(3)

G. there is a natural isomorphism $\Xi$: $\dfrac{\mathbf{C}_{total}(\mathrm{TX,Y})}{\mathbf{C}_{total}(\mathrm{DY,DX})}$

*Proof*:

A. $\forall\, X \in \mathbf{C}$, $\Diamond_X : X \to 1$ is in $\mathbf{C}_{total}$ because $D(\Diamond_X) : D(1) \to DX$ is monic in $\mathbf{C}$ (recognizing that $D(1) \cong A$)

B. $D(k) \in \mathbf{C}(DA,DX)$ is monic as DA is terminal in $\mathbf{C}$.

C. the isomorphism is $\phi$: $\frac{\mathbf{C}(DX,D1)}{\mathbf{C}(1,TX)}$, but $D1 = [1 \Rightarrow A] \cong A$.

D. $\phi^{-1}(\eta_X) = id_{DX}$, which is monic.

E. To show that the family of arrows $\{eval_X\}$ exists and is natural in X, let $D(eval_X) = \eta_{DX}$, which is monic. Then the composites $\{eval_X \circ \eta_X \in \mathbf{C}_{total}(X,X)\}$ and $\{\eta_X \circ eval_X \in \mathbf{C}_{total}(TX,TX)\}$ are natural in X, and since the only such natural families are $\{id_X\}$ and $\{id_{TX}\}$, we conclude that *eval* is a natural inverse to $\eta$ in $\mathbf{C}_{total}$.

F. $\phi^{-1}(f) = D(f) \circ \eta_{DY}$ is monic iff $D(f)$ is monic.

G. Let $\Xi(f) = D(f \circ \eta_X)$, where $f \in \mathbf{C}_{total}(TX,Y)$
and $\Xi^{-1}(D(g)) = g \circ eval_X$, where $g \in \mathbf{C}_{total}(X,Y)$
Then $\Xi^{-1}(\Xi(f)) = \Xi^{-1}(D(f \circ \eta_X)) = f \circ \eta_X \circ eval_X = f$,
and $\Xi(\Xi^{-1}(D(g)) = \Xi g \circ eval_X = D(g \circ eval_X \circ \eta_X) = D(g)$.

□

Naturality of the isomorphism $\Xi$ of Proposition 4.3.G entails four equations:

$$\Xi(f \circ D(h) = h \circ \Xi(f) \qquad \Xi(k \circ f) = \Xi(f) \circ D(k) \qquad (10a,10b)$$

$$\Xi^{-1}(D(h) \circ f) = \Xi^{-1}(f) \circ h \qquad \Xi^{-1}(f \circ k) = D(k) \circ \Xi^{-1}(f) \qquad (11a,11b)$$

Returning now to the question of the inverse of $T$, we shall show that $\tilde{T}_{X,Y} \circ T_{X,Y} = id_{TX \to TY}$ in $\mathbf{C}_{total}$. First we need a lemma.

*Lemma 4.4*: $\forall\, t : TX \forall\ c : DX$ in $\mathbf{C}_{total}$. $ans_{DX}(t,c) = ans_X(c,(eval_X\, t))$.

*Proof*: Replacing quasi-applications by compositions of arrows, the assertion of the lemma can be restated as:

$$D(c) \circ t = D(eval \circ t) \circ c : 1 \to A \qquad (12)$$

where $t : 1 \to TX$, $c : 1 \to DX$ and $TA \cong A$. Since $1 \to A \cong 1 \to TA$, it follows from (5b) that

$$\phi^{-1}(D(c) \circ t) = \phi^{-1}(t) \circ c : 1 \to A$$

However, $\phi^{-1}(t) = D(t) \circ \eta_{DX} = D(t) \circ D(eval_X) = D(eval_X \circ t)$, and for any $h : 1 \to A$,

$\phi^{-1}(h) = h$. Substitution using these two equalities gives (12).

□

*Proposition 4.5*: In $\mathbf{C}_{total}$, $\tilde{T} \circ T = id$.

*Proof*: Referring to the composition $\tilde{T} \circ T$ given in in (9), note that $\pi_1 p : \mathrm{TX}$ and $\pi_2 p : \mathrm{TY}$. Lemma 4.4 tells us that applications of these terms can be equated to

$$\pi_1 p\, \xi = \xi(eval_{\mathrm{X}}(\pi_1 p)) \tag{13a}$$

$$\pi_2 p\, \xi = \xi(eval_{\mathrm{Y}}(\pi_2 p)) \tag{13b}$$

and thus we get

$$\begin{aligned}
\tilde{T} \circ T &=_{\{(9),(13a),(13b)\}} && \lambda p.\ (\lambda c.\ \pi_1 p\, (\lambda x.\ \mathrm{K}(c\ x)(eval_{\mathrm{Y}}(\pi_2 p))), \lambda c.\ \mathrm{K}(\pi_2 p\ c)(eval_{\mathrm{X}}(\pi_1 p))) \\
&=_{\{defn.\ of\ \mathrm{K}\}} && \lambda p.\ (\lambda c.\ \pi_1 p\, (\lambda x.\ c\ x), \lambda c.\ \pi_2 p\ c) \\
&=_{\{\eta\ reduction\}} && \lambda p.\ (\lambda c.\ \pi_1 p\ c, \lambda c.\ \pi_2 p\ c) \\
&=_{\{\eta\ reduction\}} && \lambda p.\ \langle \pi_1, \pi_2 \rangle p \\
&=_{\{unicity\}} && \lambda p.\ p = id
\end{aligned}$$

□

### 4.1. Assymetric strength

The strength $T$ is not the only possibility to relate a product of computations to the computation of a product. As we have just seen, $T$ constructs the computation of a pair of values, by first reducing both components. There is an assymmetric strength

$$a_{\mathrm{T}} : \mathrm{TX} \times \mathrm{TY} \rightarrow \mathrm{T}(\mathrm{X} \times \mathrm{TY})$$

$$a_{\mathrm{T}} = \mathrm{T}(c_{\mathrm{TY,X}}) \circ t_{\mathrm{TY,X}} \circ c_{\mathrm{TX,TY}}$$

$$a_{\mathrm{T}}{}^{-1} = \sigma_{\mathrm{DX,DY}} \circ [\phi^{-1}(\bar{\pi}_1), \eta_{\mathrm{DY}}; \phi^{-1}(\bar{\pi}_2)]$$

$a_{\mathrm{T}}$ reduces only the first component of the pair it constructs, leaving the second component unaffected.

### 4.2. Projections from a lifted pair

One might also regard $\eta_{\mathrm{TX} \times \mathrm{TY}} : \mathrm{TX} \times \mathrm{TY} \rightarrow \mathrm{T}(\mathrm{TX} \times \mathrm{TY})$ as analogous to a strength, although it does not require a tensorial strength for its realization. It does, however, have a left inverse. This implies that there are projections from a 'lifted' pair, an element of $\mathrm{T}(\mathrm{TX} \times \mathrm{TY})$.

*Proposition 4.6*: In a quasi-closed category with the monad of continuations, there exists a family of arrows $\tau_{\mathrm{X,Y}} : \mathrm{T}(\mathrm{TX} \times \mathrm{TY}) \rightarrow \mathrm{TX} \times \mathrm{TY}$, natural in X and Y, such that

$\tau_{X,Y} \circ \eta_{TX \times TY} = id_{TX \times TY}$.

*Proof*: The derivation of $\tau$ is analogous to that of $\tilde{T}$ except that it uses the projections from a cartesian product, $(\pi_1)_{TX,TY}$ and $(\pi_2)_{TX,TY}$, instead of $\overline{\pi}_1$ and $\overline{\pi}_2$ to obtain

$$\tau_{X,Y} = \sigma_{DX,DY} \circ D([\phi^{-1}(\pi_1), \phi^{-1}(\pi_2)]) : T(TX \times TY) \rightarrow TX \times TY$$

or, in lambda notation

$$\tau_{X,Y} = \lambda t{:}T(TX \times TY).(\lambda c{:}DTX.t\,(c \circ \pi_1), \lambda c{:}DTY.t\,(c \circ \pi_2))$$

The composition $\tau_{X,Y} \circ \eta_{TX \times TY}$ can most easily be simplified in the lambda representation,

$$\begin{aligned}
\tau_{X,Y} \circ \eta_{TX \times TY} &= \lambda q.\, \tau_{X,Y}(\lambda c{:}D(TX \times TY).\, c\ q) \\
&= \lambda q.\, (\lambda c_X.\, \pi_1\, q\ c_X, \lambda C_Y.\, \pi_2\, q\ c_Y) \\
&= \lambda q.\, (\pi_1\, q, \pi_2\, q) \\
&= \lambda q.\, \langle \pi_1, \pi_2 \rangle q \\
&= \lambda q.\, q
\end{aligned}$$

□

**4.3. Normal objects** The unit of a monad takes a type of values into a type of its computations. In the **CAM** there are no 'unevaluated' representations for function values; each is represented as a normal term. (The same is not true of a supercombinator reducer such as the G-machine.) An element of a function-space is the involution image of a continuation transformer, and is represented in the **CAM** as a closure. (Obviously, the representations for a function value are not unique.) We can formalize the property just described.

*Definition 4.7*: In a category **C** with a strong monad, an object X is said to be a **normal object** if $\eta_X$ has a right inverse, where $\eta$ is the unit of adjunction with the Kleisli category.

□

When an abstract machine is defined by a (strong) monad of continuations on **C**, every convergent computation has a representation as an object of the subcategory **T(C)**. Every function-space object [TX⇒TY] in **C** is the image under involution of a co-exponential, interpreted as a type of continuation transformers in $\mathbf{C}^{op}$, and is a normal object.

### 4.4. The subcategory T(C) is cartesian-closed

Our goal is to show that the category of computations realized by the mechanism of continuation substitution is cartesian-closed. As T is a right adjoint functor, it carries limits in $\mathbf{C}_T$, i.e. the products and the terminal object, into limits in T(**C**). Specializing T to designate the endofunctor of the monad of continuations, since $\mathbf{C}_T$ has coexponentials and is equivalent to $\mathbf{C}^{op}$, the contravariant functor D carries these coexponentials to exponentials in **C**.

*Theorem 4.8*: If **C** is a quasi-closed bicartesian category and $(T, \eta, \mu, T)$ is the strong monad of continuations, then T(**C**) is cartesian-closed.

*Proof*:

1. **Existence of a terminal object** is assured by the bijection of hom-sets:

$$\frac{\mathbf{C}_T(IX \rightarrow \mathbf{1})}{\mathbf{C}(X \rightarrow T\mathbf{1})}$$

where the injection functor, I, is faithful and T is full.

2. **Existence and uniqueness of projections**: Let $\pi_1 : TX \times TY \rightarrow TX$ and $\pi_2 : TX \times TY \rightarrow TY$. The projections from $T(TX \times TY$ are

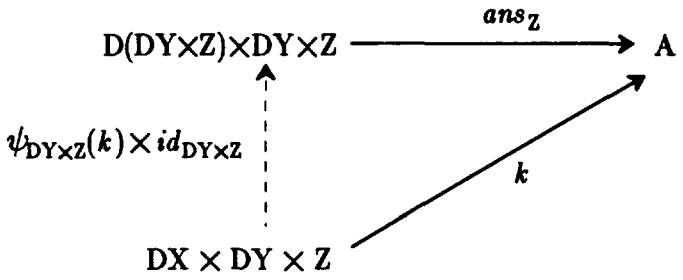
$$\pi_1 \circ \tau : T(TX \times TY) \rightarrow TX \text{ and } \pi_2 \circ \tau : T(TX \times TY) \rightarrow TY$$

These projections are unique, as

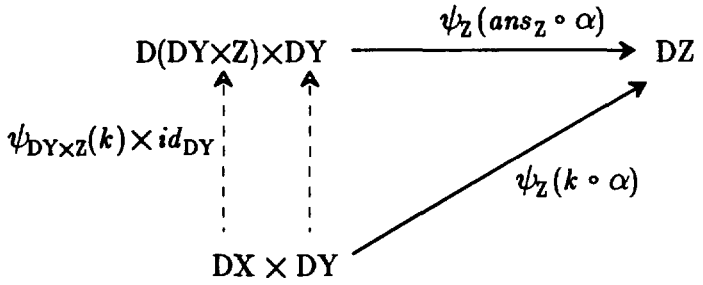
$$\eta_{TX \times TY} \circ \langle \pi_1, \pi_2 \rangle \circ \tau = id_{T(TX \times TY)}$$

The morphisms $\mathbf{fst} = \phi^{-1}(\pi_1) : DX \rightarrow D(TX \times TY)$ and $\mathbf{snd} = \phi^{-1}(\pi_2) : DY \rightarrow D(TX \times TY)$ are continuation transformers in $\mathbf{C}^{op}$ corresponding to $\pi_1 \circ \tau$ and $\pi_2 \circ \tau$ in **C**.

4. **Existence and uniqueness of exponentials**: The following diagrams illustrate the derivation of a universal arrow from $ans_Z$ by isomorphic transformation.




(a)




(b)

where $\alpha$: D(DY×Z)×(DY×Z) → (D(DY×Z)×DY)×Z is an instance of the 'administrative' isomorphism for associativity in a cartesian category. Diagram (b) expresses exponentiation of a morphism from a product, when its objects and arrows are appropriately renamed:

$$[DY \Rightarrow DZ] =_{def} D(DY \times Z)$$
$$ap_{DY,DZ} =_{def} \psi_Z(ans_Z \circ \alpha)$$
$$h = \psi_Z(k \circ \alpha)$$
$$h^{\#} = \psi_{DY\times Z}(k)$$

in which $(\_)^{\#}$ is notation for cartesian exponentiation, or 'currying'.

Instantiating X to DX, Y to DY and Z to DZ in the diagram above, we obtain

$$ap_{TY,TZ} = \psi_{DZ}(ans_{DZ} \circ \alpha)$$

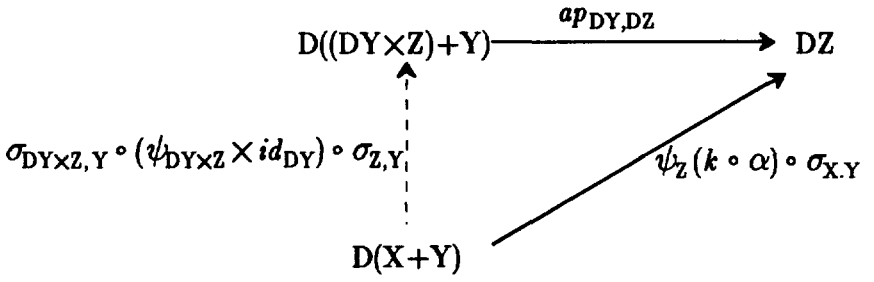
The following equation holds in **C**:

$$f^* \circ T = ap_{TY,TZ} \circ ((f^* \circ T)^{\#} \times id_{TY})$$

where $f : X \times Y \rightarrow T$ in $\mathbf{C}_T$.

□

A further, isomorphic transformation of diagram (b) in the proof above gives




This is the involution of a co-exponential diagram. The application morphism $ap_{TY,TZ}: [TY \Rightarrow TZ] \times TY \rightarrow TZ$ in **C** can also be expressed as the involution of a co-application morphism,

$$ap_{TY,TZ} = D(pa_{DZ,DY}) \circ \sigma^{-1}_{TY\times DZ,DY}$$

### 4.5. The evaluation morphisms

Finally, we can make explicit the co-unit of the adjunction between **C** and $\mathbf{C}_T$. Its Kleisli triple extension will be the application reduction morphism in T(**C**). This is

$$\overline{eval}_{TY,Z}{}^{*} = ap_{TY,TZ} \circ (eval_{1,TY\Rightarrow TZ} \times id_{TY}) \circ a_T{}^{-1}: T([TY \Rightarrow TZ] \times TY) \rightarrow TZ$$

Here, $ap_{TY,TZ} = \overline{eval}_{TY,Z}$. (Had we chosen $\tilde{T}$, rather than $a_T^{-1}$, the resulting morphism would have had the type $T([TY \Rightarrow TZ] \times Y) \rightarrow TZ$, which corresponds to call-by-value computation.) We know that this morphism is the Kleisli triple extension of a morphism $eval_{TY,Z}$ in $\mathbf{C}_T$ because T(**C**) is a full subcategory.

Function space objects are normal objects in **C**. The unit, $\eta$, instantiated for a function-space object, constructs a closure consisting of a continuation transformer and an empty environment, $\eta_{TY\Rightarrow TZ} = C_{TY\Rightarrow TZ} \circ E_{TY\Rightarrow TZ}$. In lambda notation, this is $\eta_{TY\Rightarrow TZ} = \lambda x.\ (),x$, where $(): \mathbf{1}$. Its inverse is $eval_{1,TY\Rightarrow TZ} = (\pi_1)_{TY\Rightarrow TZ,1}$. Thus

$$((\pi_1)_{TY \Rightarrow TZ, 1} \times id_{TY}) \circ a_T^{-1} : T([TY \Rightarrow TZ] \times TY) \rightarrow [TY \Rightarrow TZ] \times TY$$

is an isomorphism that relates the closure representation of a computation to an explicit continuation-transformer, environment pair. What then, is TZ in $\mathbf{C}_T$? It is the image of type TZ closures, an object formed by identifying (for all Y) the objects $[TY \Rightarrow TZ] \times TY$ in $\mathbf{C}$ as they are carried into $\mathbf{C}_T$ by the embedding map, I. The co-unit, $\epsilon$, of the adjunction (I,T) between $\mathbf{C}$ and $\mathbf{C}_T$ is the family of universal arrows $\{eval_{\_,Z} : TZ \rightarrow Z\}$ in $\mathbf{C}_T$.

## 5. Specifying the architecture of an abstract machine

Within the categorical framework developed in the preceding sections, it is now possible to define the promised machine model, the **CAM**. Its architecture can be specified by a composition of five monads, which correspond to distinct functional capabilities of the abstract machine. Composition of monads has been proposed by Moggi [21] as a technique to provide modular building blocks for programming language semantics.

For each monad (expressed here as a Kleisli triple), $(T, \eta, (\_)^*)$, and for each arrow $f : X \rightarrow TY$ in $\mathbf{C}$, we say that $f$ is *proper* for the monad[5] if $f = \eta_Y \circ g$, for some $g : X \rightarrow Y$. The proper morphisms are those that are unexceptional for the monad. For example, in the Boolean monad, whose definition in Kleisli triples is

$$\begin{aligned} BX &= X+X \\ \eta^B &= inl \\ f^*(inl\ x) &= f\ x \\ [] \ f^*(inr\ y) &= inr\ y, \end{aligned}$$

a morphism $f : X \rightarrow Y+Y$ is proper if $f = inl \circ g$ for some $g : X \rightarrow Y$.

For a proper morphism, $f = \eta \circ g$, the Kleisli composition can be replaced by ordinary composition in $\mathbf{C}$ with $f$ replaced by $g$,

$$h^* \circ f = h \circ g$$

Thus to define a proper morphism of a monad, it is not necessary to describe explicitly the action on the object structure entailed by the monad map, whereas for non-proper morphisms, its action must be specified explicitly. This allows considerable economy of

---

[5] This notion is also due to Moggi, who uses the terminology 'existing' morphism. We prefer the word 'proper', as it better conveys the sense of the classification.

notation in defining the architecture of an abstract machine, since most of its morphisms (i.e. instructions) will be proper for at least some of its constituent monads.

### 5.1. Monad construction

Moggi [21] advocates a modular approach to the construction of a categorical semantics for a programming language. In this approach, semantic domains are categories with monads. The monads may have additional structure, that is, may come equipped with additional operations that are needed to explain particular aspects of the language. He shows how categories with the required monads can be built by applying *monad constructors*, adding one feature at a time to the language being defined. Without going into the general theory behind monad construction, we shall make use of a few special instances of monad constructors to build an abstract machine.

#### 5.1.1. Composing monads

Monads impose structure upon a category. Composite structure can be expressed by a composition of monads, but monad composition is not universal. One must check, in each case, that a postulated composition actually forms a monad, i.e. that the unit and star extension exist and satisfy the monad equations. A composition might fail if, for instance, the composite object mapping did not preserve the internalization of certain morphisms as objects of the category.

If $(T_1, \eta^{T_1}, (\_)^{*1})$ and $(T_2, \eta^{T_2}, (\_)^{*2})$ are monads, we can ask whether there is a natural transformation $\tau\colon T_1T_2 \dot{\rightarrow} T_3$ such that $(T_3, \eta^{T_3}, (\_)^{*3})$ is a monad over $\mathbf{C}$ and $\tau$ has a left inverse. Naturality requires that

$$\eta_X^{T_3} = \tau_X \circ \eta_{T_2X}^{T_1} \circ \eta_X^{T_2}$$

The requirement that $\tau$ has a left inverse is to ensure that the object mapping $T_3$ does not identify objects that are not identified by $T_1T_2$.

When the above conditions are satisfied, we call the result the *monad composition* of $T_1$ with $T_2$ by $\tau$.

#### 5.1.2. An example: state transformers

Consider the monad of *state transformers*,

$$S_T X = S \Rightarrow X \times S$$

$$\eta = \lambda x.\, \lambda s.\, x, s$$

$$f^* = \lambda t.\, \lambda s.\, \textbf{let}\ (x,s') = t\ s\ \textbf{in}\ f\ x\ s'$$

This monad has previously been interpreted as computations that produce side effects on a store [23, 26].

If $(\mathrm{T}, \eta, (\_)^{*\mathrm{T}})$ is a monad to be composed with $\mathrm{S}_\mathrm{T}$, the required natural transformation has the typing

$$\tau_\mathrm{X} : (\mathrm{S} \Rightarrow \mathrm{TX} \times \mathrm{S}) \to \mathrm{S} \Rightarrow \mathrm{T}(\mathrm{X} \times \mathrm{S})$$

The composition of $\mathrm{S}_\mathrm{T}$ with T by $\tau$ is given by

$$\mathrm{S}_\mathrm{T} \cdot \mathrm{TX} = \mathrm{S} \Rightarrow \mathrm{T}(\mathrm{X} \times \mathrm{S})$$

$$\eta_\mathrm{X}^{\mathrm{S}_\mathrm{T}\mathrm{T}} = \lambda x.\, \tau \circ (\eta_{\mathrm{TX}}^{\mathrm{S}_\mathrm{T}}(\eta_\mathrm{X}^\mathrm{T}\, x))$$

$$f^{*\mathrm{ST}} = \lambda u.\, \lambda s.\, \textbf{let}\ (\xi, s') = \tau^{-1} u\ s$$
$$\textbf{in}\ f^{*\mathrm{T}}\, \xi\, s'$$

where $f : \mathrm{X} \to \mathrm{S} \Rightarrow \mathrm{T}(\mathrm{Y} \times \mathrm{S})$ and $f^{*\mathrm{T}} : \mathrm{TX} \to \mathrm{S} \Rightarrow \mathrm{T}(\mathrm{Y} \times \mathrm{S})$.

In particular, $\mathrm{S}_\mathrm{T}$ can be composed with another monad of state transformers. The required natural transformation is:

$$\tau_\mathrm{X} : (\mathrm{S} \Rightarrow (\mathrm{R} \Rightarrow \mathrm{X} \times \mathrm{R}) \times \mathrm{S}) \to \mathrm{S} \Rightarrow \mathrm{R} \Rightarrow (\mathrm{X} \times \mathrm{S}) \times \mathrm{R}$$

$$\tau = \lambda\sigma.\lambda s.\, \lambda r.\, \textbf{let}\ \xi, s' = \sigma\, s$$
$$\textbf{in let}\ x, r' = \xi\, r$$
$$\textbf{in}\ (x, s'), r'$$

$$\tau^{-1} = \lambda\nu.\lambda s.\textbf{let}\ \varsigma = \nu\, s$$
$$\textbf{in}\ \langle \pi_1 \circ \pi_1, \pi_2 \rangle \circ \varsigma, (\pi_2 \circ \pi_1)(\varsigma())$$

Notice that $\tau^{-1}$ is not a right inverse to $\tau$. Another useful example is the composition of $\mathrm{S}_\mathrm{T}$ with the monad of continuations. The required natural transformation is:

$$\tau_\mathrm{X} : (\mathrm{S} \Rightarrow ((\mathrm{X} \Rightarrow \mathrm{A}) \Rightarrow \mathrm{A}) \times \mathrm{S}) \to \mathrm{S} \Rightarrow ((\mathrm{X} \times \mathrm{S}) \Rightarrow \mathrm{A}) \Rightarrow \mathrm{A}$$

$$\tau = \lambda\sigma.\lambda s.\, \lambda c_2.\textbf{let}\ c_1, s' = \sigma\, s$$
$$\textbf{in}\ c_1(\lambda x.c_2(x, s'))$$

$$\tau^{-1} = \lambda\nu.\lambda s.\, \textbf{let}\ t_2 = \nu\, s$$
$$\textbf{in}\ (\lambda c_1.t_2(\lambda(x,s').c_1\, x)), t_2(\lambda(x,s').s')$$

## 5.2. The categorical abstract machine

The **CAM** is a composition of a series of state transformer monads with a monad of continuations. The state transformers each contribute one more component to the machine state. Instructions of the machine form groups that utilize successively wider views of the machine state. The action of the machine is modeled as a continuation, applied to a machine state.

The state space of the **CAM** will be defined 'inside out'. Starting from a type X of value representations, we first augment it with a state transformer Reg. Its state object is the type of contents of a register file, $Rec(X)$, where $Rec$ is a recursively defined type,

$$Rec(R) = R \times Rec(R).$$

An element of this type serves as a stack of intermediate value representations used in a computation.

At the second level, the **CAM** has a state transformer Cc, which adds another state object **2** that corresponds to a one-bit condition-code register. At the third level, the state transformer Cs, adds a state object that is a type of control stores. A control store will be modeled as a list of instructions. At the fourth level is the state transformer M, whose state object is the type of a stack of contexts. The fifth level specifies a monad of continuations, T.

The entire machine is then the monad composition

$$\mathbf{CAM} = \mathrm{Reg}(\mathrm{Cc}(\mathrm{Cs}(\mathrm{M}(\mathrm{T}\cdot\mathrm{Id}))))$$

where Id is the identity monad, here used as a placeholder for an object variable.

## 5.3. Using a control store to realize continuations

The state object of Cs is $List(\mathrm{Control})$, where $\mathrm{Control} = \mathrm{S} \rightarrow \mathrm{TS}$ and

$$\mathrm{S} = Rec(\mathrm{X}) \times \mathbf{2} \times List(\mathrm{Control}) \times Rec(\mathrm{Context})$$

S is the state object for M and the type of the state of the entire machine. A machine continuation has the type

$$\mathrm{Mcont} = \mathrm{S} \Rightarrow \mathrm{A}$$

and a context has the type

$$\text{Context} = Rec(\text{X}) \times \mathbf{2} \times List(\text{Control}) \times \text{Mcont}$$

A context element is formed of a state element, less the current value component, and with a machine continuation appended. A context object is isomorphic to $\text{X} \Rightarrow \text{A}$, the type of value continuations.

A single step of machine operation is represented by[6]

$$Step : \text{Mcont} \rightarrow \text{Mcont}$$

$$Step\ m \left( \frac{\rho, b, op;s}{C} \right) = op \left( \frac{\rho, b, s}{C} \right) m$$

$$Step\ m \left( \frac{\rho, b, []}{C} \right) = m \left( \frac{\rho, b, []}{C} \right)$$

where ';' is the infix list constructor and [] designates an empty list. If *op* is proper for M, then it has the form $\lambda t.\, \lambda m.\ m\,(op'\, t)$, where $op' : \text{S} \rightarrow \text{S}$. As a morphism of **C**, an instruction proper for M is

$$op = ans_{\text{S}} \circ C_{\text{S,DS}} \circ (op' \times id_{\text{DS}}) \circ \eta^{\text{ST}}$$

The only instructions of **CAM** that are non-proper for M are 'call', 'ret' and 'eval'.

The machine executes by performing *Step* repeatedly, beginning with the components of an initial machine state. That is,

$$Run = \text{Y}(Step) = \lim_{i \rightarrow \infty} Step^i(\perp_{\text{Mcont}})$$

where Mcont is a pointed cpo and $\perp_{\text{Mcont}}$ is its least element. An execution with program store $s_0$ from initial data $x$ is $Run \left( \frac{(x, \rho_0), tt, s_0}{C_0} \right)$, where $\rho_0$ is a constant of $Rec(\text{X})$ and $C_0$ is a constant of $Rec(\text{Context})$. A sequence of steps, $Step^i$, applied to a machine state yields a sequence of $i$ instructions that comprise a partial elaboration of the continuation from the given machine state. For example, if the initial instruction of the control store happens to be proper for Cs, $op = \lambda\sigma.\lambda m.m\,(op'\,\sigma)$, where $op' : \text{S} \rightarrow \text{S}$, then

---

[6] In displaying patterns, we shall let $\rho$ range over $Rec(\text{X})$ and $s$ range over $List(\text{Control})$. The notation is similar to conventions often used in register-transfer descriptions of machines.

$$Step\ m\ \left[\frac{\rho, b, op;s}{C}\right] = m\,(op'\left[\frac{\rho, b, s}{C}\right]) = (\phi^{-1}(op);m)\left[\frac{\rho, b, s}{C}\right] = Step\ (\phi^{-1}(op);m)\left[\frac{\rho, b, s}{C}\right]$$

The initial program store, $s_0$, is actually a free variable of the expression of an abstract machine, as it also occurs in the definitions of some control transfer instructions.

This definition of machine semantics separates the definition of a computation from the question of whether it terminates. A computation terminates if it reaches a fixed point after a finite number of steps.

Instructions proper for Cs preserve the control store, while non-proper instructions may replace the current control with another one. We introduce the list function

$$\begin{aligned} &indx\,(0)\,s = s \\ &indx\,(n+1)(c;s) = indx\,(n)\,s \end{aligned}$$

which allows non-negative integers to be used as labels in control stores. When $indx(l)$ is composed with $\mathrm{K}\,s_0 = \lambda s.s_0$, which replaces the current control store with a constant (the initial control store), we obtain a composite function $indx(l) \circ \mathrm{K}\,s_0$, giving the effect of a control transfer directed to a label, $l$. The fixpoint computation, *Run*, makes use of the indexable control store in an essential way. Each time an instruction directs control to a label, the control store is effectively re-initialized to the sequence beginning at the specified label. This abstract machine model is capable of evaluating recursively defined functions. In order that it could compute fixpoints of values not of a functional type, a natural mechanism to add to the machine would be an addressable data store. We have not made that extension in the present paper.

### 5.4. Instructions of the CAM

We begin by giving the instructions that correspond to proper morphisms for the composite monad **CAM**. These instructions only affect the current value representation; they make no use of the added components of machine state. Accordingly, some instructions (the arithmetic instructions, here) require auxiliary definition to make them precise.

The notation is similar to the register-transfer descriptions often used to describe the architecture of concrete machines. The sense of the arrows corresponds to arrows in the Kleisli category $\mathbf{C}_\mathbf{T}$, induced by the monad of continuations. This is the representation we find most intuitive. The instructions themselves compose as con-

tinuation transformers in **C**.

**Instructions proper for all monads**

| | |
|---|---|
| $\text{quote}_X(z)$ | $x \rightarrow z$ |
| fst | $(x,y) \rightarrow x$ |
| snd | $(x,y) \rightarrow y$ |
| add | $(x,y) \rightarrow x+y$ |
| neg | $x \rightarrow -x$ |

There could, of course be additional proper instructions. We shall make no further use of the arithmetic instructions, which have been included only as examples.

**Instructions non-proper for Reg**

The notation we shall use for the **CAM** register file will usually make explicit its first element, called the *Term Register* in the original **CAM** [3], and the rest of the registers, called the *Stack*.

| | |
|---|---|
| push | $x,\rho \rightarrow x,(x,\rho)$ |
| pop | $x,(y,\rho) \rightarrow x,\rho$ |
| swap | $x,(y,\rho) \rightarrow y,(x,\rho)$ |
| mk_pr | $y,(x,\rho) \rightarrow (x,y),\rho$ |

The instructions 'push' and 'pop' correspond to **dupl** and **drop** of the Linear Abstract Machine [14]. The instruction 'swap' is the exchange morphism of a symmetric product. 'mk_pr' is an instance of the associativity morphism, $\alpha$.

**Instructions non-proper for Cc**

| | | |
|---|---|---|
| eq0 | $(x,\rho),b \rightarrow (x,\rho),tt$ | if $0=x \in X$ |
| eq0 | $(x,\rho),b \rightarrow (x,\rho),ff$ | otherwise |

In a hardware realization, this instruction could set a condition code register.

**Instructions non-proper for Cs**

The next set of instructions are those that transfer control. Recall that the state object for Cs is *List*(Control), an object whose elements represent control stores. In the descriptions below, $\rho$ ranges over environment objects and $s$ ranges over control stores.

| | |
|---|---|
| jmp($l$) | $\rho,b,s \rightarrow \rho,b,indx(l)\,s_0$ |
| jfalse($l$) | $\rho,b,s \rightarrow$ **if** $b=tt$ **then** $\rho,b,s$ **else** $\rho,tt,indx(l)\,s_0$ |

$$\text{stop} \qquad \rho, b, s \longrightarrow \rho, b, \text{stop}; s$$

**Instructions non-proper for M**

This brings us to the most complex operations, those non-proper for M. Here the notation for an object is extended to include stacked contexts.

$$\text{call}(l) \qquad \frac{(x,\rho), b, s}{C} \longrightarrow \lambda m.Run\left(\frac{(x,\rho_0), tt, indx(l)\, s_0}{\dfrac{\rho, b, s, m}{C}}\right)$$

$$\text{ret} \qquad \frac{(x,\rho), b, s}{\dfrac{\rho', b', s', m'}{C}} \longrightarrow \lambda m.m'\left(\frac{(x,\rho'), b', s'}{C}\right)$$

$$\text{eval} \qquad \frac{((l,x),\rho), b, s}{C} \longrightarrow \lambda m.Run\left(\frac{(x,\rho_0), tt, indx(l)\, s_0}{\dfrac{\rho, b, s, m}{C}}\right)$$

These non-proper instructions can be identified with the categorical morphisms that underlie them. In general, for an instruction $op : \mathrm{S} \longrightarrow \mathrm{TS}$, we need to express its Kleisli triple extension, $op^* = \mathrm{D}(\phi^{-1}(op))$. Under involution, the instruction can be represented as a continuation transformer with the form

$$\phi^{-1}(op) = pa_{\mathrm{DS}}; (\varsigma_{op} + id_{\mathrm{DS}}); \mathrm{DS} \Rightarrow \mathrm{A} + \mathrm{DS}$$

in the category $\mathbf{C}^{\mathrm{op}}$, where $pa_{\mathrm{DS}} : \mathrm{DS} \longrightarrow (\mathrm{TS} \times \mathrm{DS}) + \mathrm{DS}$. Note that $\mathrm{TS} \times \mathrm{DS} = [\mathrm{DS} \Leftarrow \mathrm{DS}]$ is a co-exponential object in this category, and that $pa_{\mathrm{DS}}$ realizes continuation substitution. Then

$$op^* = \mathrm{D}(pa_{\mathrm{DS}}) \circ (\mathrm{D}(\varsigma_{op}) \times id_{\mathrm{DS}}) : \mathrm{DA} \times \mathrm{TS} \longrightarrow \mathrm{TS}$$

Recall that $\mathrm{DA} = \mathbf{1}$ and thus the type is isomorphic to $\mathrm{TS} \longrightarrow \mathrm{TS}$.

If $op$ is proper for M, $op = \lambda\sigma.\lambda m.m(op'\sigma)$, then

$$\varsigma_{op} = ans_{\mathrm{S}} \circ C_{\mathrm{S,DS}} \circ \langle op' \circ \pi_1, \pi_2 \rangle : \mathrm{TS} \times \mathrm{DS} \Rightarrow \mathrm{A}$$

For a uniformly terminating proper instruction, the type of $\phi^{-1}(op)$ is isomorphic to $\mathrm{DS} \longrightarrow \mathrm{DS}$ and can be represented in lambda notation as $\lambda m.op'; m$.

For a non-proper instruction, $\varsigma_{op}$ is more complex. For instance,

$$\varsigma_{\mathrm{call}(l)} = \mathrm{Run} \circ \xi_{\mathrm{call}(l)}$$

where $\xi_{\mathrm{call}(l)}$ is given using variables rather than compositions of projection morphisms for better readability:

$$\xi_{\mathrm{call}(l)}\left(\left\langle\frac{(x,\rho),b,s}{C}\right\rangle,m\right)=\left\langle\frac{(x,()),tt,indx(l)\,s_0}{\frac{\rho,b,s,m}{C}}\right\rangle$$

The operational effect of 'call($l$)' is to substitute the continuation represented by the label $l$ in place of the current continuation, which is saved. The categorical operation that corresponds to 'saving' a continuation is co-application of a continuation abstraction.

The return instruction is represented in analogous fashion,

$$\varsigma_{\mathrm{ret}} = ans_{\mathrm{S}} \circ C_{\mathrm{S,DS}} \circ \xi_{\mathrm{ret}}$$

where

$$\xi_{\mathrm{ret}}\left(\left\langle\frac{(x,\rho),b,s}{\frac{\rho',b',s',m'}{C}}\right\rangle m\right)=\left(\left\langle\frac{(x,\rho'),b',s'}{C}\right\rangle,m'\right)$$

'eval' corresponds to $\epsilon$, the co-unit of the adjunction between **C** and $\mathbf{C}_{\mathrm{T}}$. Its representation is similar to that of 'call', except that the label argument to *indx* is gotten by taking the first projection of the value component of the state.

## 6. Translating lambda calculus for the CAM

The deBruijn combinatory calculus can be compiled for the **CAM** by a compilation scheme **C** that translates expressions without evaluation. For this translation, we do not require the full machine. We use the *Basic* abstract machine, **BCAM**, which is similar to **CAM** but omits the condition code component of machine state, **2**, and the instructions 'add', 'neg', 'pop', 'eq0', 'jmp', 'jfalse', 'stop' and 'call'. The compilation scheme is:

$$\mathbf{C}\lfloor\overline{0}\rfloor = \mathrm{snd}$$

$$\mathbf{C}\lfloor\overline{n+1}\rfloor = \mathrm{fst};\mathbf{C}\lfloor\overline{n}\rfloor$$

$$\mathbf{C}\lfloor M\,N\rfloor = \mathrm{push};\mathbf{C}\lfloor N\rfloor;\mathrm{swap};\mathbf{C}\lfloor M\rfloor;T$$

$$\mathbf{C}\lfloor\Lambda M\rfloor = \mathrm{push};\mathrm{quote}(l);\mathrm{swap};\mathrm{mk\_pr}$$

where 'label($l$);$\mathbf{C}\lfloor M\rfloor$;eval;ret'
is appended to the control store

In this translation, $T$ is a macro instruction,

$$T = \text{push};\mathit{Rot};\text{snd};\text{swap};\text{mk\_pr};\text{swap};\text{fst};\text{swap};\text{mk\_pr}$$

*Rot* is another macro-instruction that exchanges the two values nearest the top of the **BCAM** register file.

$$\mathit{Rot} = \text{mk\_pr};\text{mk\_pr};\text{push};\text{snd};\text{fst};\text{swap};\text{push};\text{fst};\text{swap};\text{snd};\text{snd};$$

### 6.1. Coherence of the implementation

Coherence of a model with a language is the property that equivalence of terms in the language implies equivalence of their interpretations in the model. In this section, we establish that the interpretation given by the translation scheme **C** is coherent with the equivalence defined by $\beta$-conversion of terms not protected by $\Lambda$. This is equivalent to asserting that programs executed by **BCAM** compute (weak) head-normal forms of the $\lambda_\beta$-calculus.

Equivalence in the model must be extensional equivalence of functions, since terms of the language are interpreted in a function space. We designate this relation by '$\equiv$', and define it to be the least relation on **BCAM** terms that is reflective, symmetric, closed under transitivity, and satisfies

$$\forall\, e,\rho,C.\ \frac{e,\rho,\mathbf{C}\lfloor M\rfloor;\text{eval}}{C} = \frac{e,\rho,\mathbf{C}\lfloor N\rfloor;\text{eval}}{C} \Rightarrow \mathbf{C}\lfloor M\rfloor \equiv \mathbf{C}\lfloor N\rfloor$$

The proof of coherence relies upon several technical lemmas, some of which are of interest in their own right.

*Lemma 6.1*:

$$\forall\, M.\ \forall\, s,e,\rho,C.\ \exists\, !e'.\ \frac{e,\rho,\mathbf{C}\lfloor M\rfloor;s}{C} = \frac{e',\rho,s}{C}$$

*Proof*: We show by induction on the structure of $M$ that $\mathbf{C}\lfloor M\rfloor$ is proper for **BCAM**. Recall that an instruction proper for M can affect only the value of the Term Register component, leaving all other components of state unchanged.

$M = \overline{n}$:

by induction on $n$. $\mathbf{C}\lfloor\overline{0}\rfloor = \text{snd}$, which is proper. $\mathbf{C}\lfloor\overline{n+1}\rfloor = \text{fst};\mathbf{C}\lfloor\overline{n}\rfloor$, which is the composition of a proper instruction with a sequence proper by hypothesis.

$M = M'N'$:

Examine the definition of $\mathbf{C}\lfloor M'N'\rfloor$. There is no occurrence of 'eval' or 'ret', the instructions non-proper for M and Cs. The instructions non-proper for Reg are

'push', 'swap', and 'mk_pr'. But a sequence of these instructions remains proper for Reg provided that (1) every occurrence of 'push' is matched (exactly) by a following occurrence of 'mk_pr', and (2) every occurrence of 'swap' is bracketed between an occurrence of 'push' and one of 'mk_pr'. Proper instructions may be freely interleaved in a sequence without affecting its propriety.

Recoding the instruction sequence for $\mathbf{C}\lfloor M'N' \rfloor$ with the macro instructions $T$ and *Rot* expanded, proper instructions replaced by _, 'push' by '$\langle$', 'swap' by $s$ and 'mk_pr' by '$\rangle$', gives

$$\langle \mathbf{C}\lfloor N' \rfloor s\, \mathbf{C}\lfloor M' \rfloor \langle\rangle\rangle\langle \_\_ s \langle \_ s \_\_\_ s \rangle s \_ s \rangle$$

The embedded instruction sequences $\mathbf{C}\lfloor M' \rfloor$ and $\mathbf{C}\lfloor N' \rfloor$ are proper by hypothesis. The sequence can be seen to satisfy conditions (1) and (2).

$M = \Lambda M'$

Again, the instruction sequence for $\mathbf{C}\lfloor \Lambda M' \rfloor$ contains no occurrence of 'eval' or 'ret'. Its recoding is $\langle \_ s \rangle$, which satisfies conditions (1) and (2) above, hence the sequence is proper.

Uniqueness of the term $e'$ follows from the induction ($\mathbf{C}\lfloor M \rfloor$ is finite) and the fact that each instruction is interpreted by a (total) function.

□

*Notation*: In what follows, we shall write $\mathbf{C}\lfloor M \rfloor e$ to stand for the unique term $e'$ asserted by Lemma 6.1.

*Lemma 6.2*: If a label, $l$, has a binding in the initial control store

$$l\colon \mathbf{C}\lfloor M \rfloor; \mathrm{eval}; \mathrm{ret}; \cdots$$

then for evaluations that terminate,

$$\forall\, s,e,\rho,C.\ (1)\quad \exists\, !e'.\ \frac{(l,e),\rho,\mathrm{eval};s}{C} = \frac{e',\rho,s}{C}$$

$$(2)\quad \frac{(l,e),\rho,\mathrm{eval};s}{C} = \frac{\mathbf{C}\lfloor M \rfloor e,\rho,\mathrm{eval};s}{C}$$

$$(3)\quad \forall\, e',s'.\ \frac{e,\rho,\mathbf{C}\lfloor M \rfloor;\mathrm{eval};\mathrm{ret};s'}{\frac{(e',\rho):s}{C}} = \frac{e,\rho,\mathbf{C}\lfloor M \rfloor;\mathrm{eval};s}{C}$$

(The term $e'$ asserted in (1) will be designated 'eval($\mathbf{C}\lfloor M \rfloor e$)' when it is referred to in the derivation of (3) in the proof.)

*Proof*: is by a well-founded induction based upon the depth of the stacked contexts. We show that if the conclusions of the lemma hold at states of the computation where the stack is more deeply nested, then they also hold at states where the stack is shallower. The stack depth is finite at every step of a terminating computation.

(1)
$$\frac{(l,e),\rho,\text{eval};s}{C}$$
$$=_{\{\text{eval}\}} \quad \frac{e,\rho,\mathbf{C}\lfloor M\rfloor;\text{eval};\text{ret};\cdots}{\frac{((l,e),\rho):s}{C}} \qquad (*)$$
$$=_{\{\textit{Lemma 6.1}\}} \frac{\mathbf{C}\lfloor M\rfloor e,\rho,\text{eval};\text{ret};\cdots}{\frac{((l,e),\rho):s}{C}}$$
$$=_{\{\textit{Hypoth.}(2)\}} \frac{(l,e),\rho,\text{eval};\text{ret};\cdots}{\frac{((l,e),\rho):s}{C}}$$
$$=_{\{\textit{Hypoth.}(1)\}} \frac{e',\rho,\text{eval};\text{ret};\cdots}{\frac{((l,e),\rho):s}{C}} K\ \text{ret};\cdots$$
$$=_{\{\text{ret}\}} \quad \frac{e',\rho,s}{C}$$

(2) From (*) above, the next step is:
$$=_{\{\textit{Hypoth.}(3)\}} \frac{e,\rho,\mathbf{C}\lfloor M\rfloor;\text{eval};s}{C}$$
$$=_{\{\textit{Lemma 6.1}\}} \frac{\mathbf{C}\lfloor M\rfloor e,\rho,\text{eval};s}{C}$$

(3)
$$\frac{e,\rho,\mathbf{C}\lfloor M\rfloor;\text{eval};\text{ret};\cdots}{\frac{e',\rho:s}{C}}$$
$$=_{\{\textit{Lemma 6.1}\}} \frac{\mathbf{C}\lfloor M\rfloor e,\rho,\text{eval};\text{ret};\cdots}{\frac{e',\rho:s}{C}}$$
$$=_{\{\textit{Hypoth. (2)}\}} \frac{(l,e),\rho,\text{eval};\text{ret};\cdots}{\frac{e',\rho:s}{C}}$$
$$=_{\{\textit{Hypoth. (1)}\}} \frac{\text{eval}(\mathbf{C}\lfloor M\rfloor e),\rho,\text{ret};\cdots}{\frac{e',\rho:s}{C}}$$

$$=_{\{\text{ret}\}} \quad \frac{\text{eval}(\mathbf{C}\lfloor M\rfloor e),\rho,s}{C}$$

$$=_{\left\{\begin{array}{l}\textit{Hypoth.}(1),\\ \textit{Lemma 6.1}\end{array}\right\}} \frac{e,\rho,\mathbf{C}\lfloor M\rfloor;\text{eval};s}{C}$$

The uses of the inductive hypotheses in the derivations can be indexed by the level of stacked contexts at the point that each hypothesis is invoked:

to derive H1: H2(1), H1(1)
to derive H2: H3(0)
to derive H3: H2(1), H1(1), H1(0)

from which it can be seen that the induction is well-founded.

□

*Corollary 6.3*: $\frac{e,(e',\rho)}{C}:\mathbf{C}\lfloor \Lambda M\rfloor;T;\cdots \equiv \frac{(e,e'),\rho}{C}:\mathbf{C}\lfloor M\rfloor;\cdots.$

*Proof*:

$$\frac{e,(e',\rho),\mathbf{C}\lfloor \Lambda M\rfloor;T;\text{eval};\cdots}{C}$$

$$=_{\{\textbf{C-}\textit{translation}\}} \frac{e,(e',\rho),\text{push};\text{quote}(l);\text{swap};\text{mk\_pr};T;\text{eval};\cdots}{C}$$

where $l:\mathbf{C}\lfloor M\rfloor;\text{eval};\text{ret}\cdots$

$$=_{\{\textit{Lemma 6.2.1}\}} \frac{(l,e),(e',\rho),T;\text{eval};\cdots}{C}$$

$$=_{\{T\}} \quad \frac{(l,(e,e')),\rho,\text{eval};\cdots}{C}$$

$$=_{\{\textit{Lemma 6.2.1}\}} \frac{(e,e'),\rho,\mathbf{C}\lfloor M\rfloor;\text{eval};\cdots}{C}$$

□

In the following, we make use of the deBruijn substitution convention. Our notation is adapted from [11]. The notation $M\{N\}_i$ designates the substitution of $N$ for all occurrences of the $i^{\text{th}}$ bound variable in $M$, where the index $i$ counts the binding-height of each variable occurrence in a term. The rules for substitution are given inductively on the structure of the term $M$ in the proof of Lemma 6.4, below. The definition depends upon an auxiliary function that calculates index renormalization in the substituted term, $N$. Its definition is:

$$R_i^j(\overline{m}) = \begin{cases}\overline{m} & \text{if } m<j\\ \overline{m+i} & \text{if } m\geq j\end{cases}$$

$$R_i^j(M'N') = R_i^j(M')R_i^j(N')$$
$$R_i^j(\Lambda M) = \Lambda(R_i^{j+1}(M))$$

*Lemma 6.4*: $\mathbf{C}\lfloor M\{N\}_i \rfloor(...(e, e_{i-1})...e_0) \equiv \mathbf{C}\lfloor M \rfloor(...((e, \mathbf{C}\lfloor N \rfloor e), e_{i-1})...e_0)$

*Proof*: by well-founded induction on the term structure and the substitution index, $i$.

$M = \overline{m}$ and $m < i$

$\overline{m}\{N\}_i = \overline{m}$,

$\mathbf{C}\lfloor \overline{m} \rfloor(...(e, e_{i-1})...e_0) = e_m = \mathbf{C}\lfloor \overline{m} \rfloor(...((e, \mathbf{C}\lfloor N \rfloor e), e_{i-1})...e_0)$

$M = \overline{m}$ and $m = i$

$\overline{i}\{N\}_i = R_i^0(N)$

$\mathbf{C}\lfloor R_i^0(N) \rfloor(...(e, e_{i-1})...e_0) = \mathbf{C}\lfloor N \rfloor e = \mathbf{C}\lfloor \overline{i} \rfloor(...((e, \mathbf{C}\lfloor N \rfloor e), e_{i-1})...e_0)$

$M = \overline{m}$ and $m > i$

$\overline{m}\{N\}_i = \overline{m-1}$

$\mathbf{C}\lfloor \overline{m-1} \rfloor(...(e, e_{i-1})...e_0) = \mathbf{C}\lfloor \overline{m-1-i} \rfloor e = \mathbf{C}\lfloor \overline{m} \rfloor(...((e, \mathbf{C}\lfloor N \rfloor e), e_{i-1})...e_0)$

$M = M'N'$

$(M'N')\{N\}_i = M'\{N\}_i N'\{N\}_i$

$$\begin{aligned}
& \mathbf{C}\lfloor (M'N')\{N\}_i \rfloor(...(e, e_{i-1})...e_0), \rho : \cdots \\
=_{\{\textit{Lemma 6.1, C-translation}\}} & (...(e, e_{i-1})...e_0), \rho : \text{push}; \mathbf{C}\lfloor N'\{N\}_i \rfloor; \text{swap}; \mathbf{C}\lfloor M'\{N\}_i \rfloor; T \cdots \\
=_{\{\textit{Lemma 6.1, C-translation}\}} & \mathbf{C}\lfloor M'\{N\}_i \rfloor(...(e, e_{i-1})...e_0), (\mathbf{C}\lfloor N'\{N\}_i \rfloor(...(e, e_{i-1})...e_0), \rho) : T \cdots \\
\equiv_{\{\textit{Hypothesis}\}} & \mathbf{C}\lfloor M' \rfloor(...((e, \mathbf{C}\lfloor N \rfloor e), e_{i-1})...e_0), (\mathbf{C}\lfloor N' \rfloor(...((e, \mathbf{C}\lfloor N \rfloor e), e_{i-1})...e_0), \rho) : T \cdots \\
=_{\{\textit{C-translation, Lemma 6.1}\}} & \mathbf{C}\lfloor M'N' \rfloor(...((e, \mathbf{C}\lfloor N \rfloor e), e_{i-1})...e_0), \rho : \cdots
\end{aligned}$$

$M = \Lambda M'$

$(\Lambda M')\{N\}_i = \Lambda(M'\{N\}_{i+1})$

$$\begin{aligned}
& \mathbf{C}\lfloor \Lambda(M'\{N\}_{i+1}) \rfloor(...(e, e_{i-1})...e_0), (e', \rho) : T \\
\equiv_{\{\textit{Corollary 6.3}\}} & \mathbf{C}\lfloor M'\{N\}_{i+1} \rfloor((...(e, e_{i-1})...e_0), e'), \rho \\
\equiv_{\{\textit{Hypothesis}\}} & \mathbf{C}\lfloor M' \rfloor((...((e, \mathbf{C}\lfloor N \rfloor e)\, e_{i-1})...e_0), e'), \rho \\
\equiv_{\{\textit{Corollary 6.3}\}} & \mathbf{C}\lfloor \Lambda M' \rfloor(...((e, \mathbf{C}\lfloor N \rfloor e)\, e_{i-1})...e_0), (e', \rho) : T
\end{aligned}$$

□

*Theorem 6.5*: **Coherence** ($\beta$) $\quad \mathbf{C}\lfloor(\Lambda M)N\rfloor \equiv \mathbf{C}\lfloor M\{N\}_0\rfloor$

*Proof*:

$$\begin{aligned} & \mathbf{C}\lfloor(\Lambda M)N\rfloor e \\ \equiv_{\left\{\begin{array}{l}\textit{C-translation},\\ \textit{Lemma 6.1}\end{array}\right\}} & \mathbf{C}\lfloor M\rfloor(e, \mathbf{C}\lfloor N\rfloor e) \\ \equiv_{\{\textit{Lemma 6.4}\}} & \mathbf{C}\lfloor M\{N\}_0\rfloor e \end{aligned}$$

□

## 7. Conclusions

This paper provides further evidence, if any were needed, that categories with monads provide useful models for computation. Monads characterize the additional structure required in a category to realize an abstract machine. Monad composition, using monads of state transformers and a monad of continuations, appears to be a satisfactory way to compose an abstract machine. The structure obtained in this way is remarkably similar to the architecture of machines intended for realization in hardware. This structure also has the advantage that it renders a coherence proof simpler than it might otherwise be, as most details of the proof (such as verifying that substitution for variables is correctly implemented) involve instructions that are proper for many monads of the composition. Proof of coherence is a preferred method by which to show the correctness of an implementation, as it relates a machine model directly to the logical specification of a programming language, rather than to a denotational model.

We have provided a categorical framework that allows the formal description of abstract machine models to be carried to a finer degree of detail than has previously been attempted. In particular, it supports the specification of control structure as well as the transformation of value representations. To express control structure, we have introduced objects interpreted as types of continuations and arrows interpreted as continuation transformers, in addition to functions on values. The model exploits categorical duality to relate values and continuations. Substitution of continuations, an operation dual to the substitution of values, is the interpretation we give to co-application. This is a different interpretation than has been given by Filinski in considering dual categories with closure of products and (co)closure of coproducts.

The abstract machine we have constructed is very similar to Mauny's 'lazy' version of the **CAM** [19]. Mauny proposed operations 'freeze' and 'unfreeze' whose actions are respectively, to construct a closure from a code pointer and an environment, and to evaluate a closure. In our model, the counterpart of 'freeze' is embedded in the macro-instruction $T$ and the counterpart of 'unfreeze' is 'eval'. In Mauny's compilation rules, 'unfreeze' is inserted to force evaluation of an argument when it is referenced by a strict operator, whereas in our model, 'eval' is applied to the term value returned by a function application. These two strategies are equivalent. The 'eval' instruction does not require a recursive definition when function-space objects are normal objects.

An abbreviated coherence proof has been given for the **CAM** in [4]. A detailed coherence proof for an implementation of a lazy, functional language by the G-machine was presented in David Lester's doctoral thesis [16]. His proof is based upon point-wise reasoning and uses induction on the data structures of the abstract machine, rather than a categorical semantics.

Asperti [1] has considered several variants of the **CAM** in a categorical framework based upon cartesian-closed categories. In this work the categorical model is developed in more detail than was the original model of [4], but it does not address a question we have explored here, of how the required exponentials may be fabricated in a category where less structure than cartesian closure is assumed.

The analogy between involution and various forms of logical negation is well known and has been exploited by [8] to derive typings for first-class control constructs. Involution and duality in linear categories has been studied in a detailed paper by Marti-Oliet and Meseguer [18]. There, the involution is defined by an isomorphism, analogous to the double-negation elimination rule of linear (and classical) logic.

Finally, we wish to point out some directions in which to extend this work. An obvious extension of the **CAM** definition would be to accommodate evaluation-sharing (true lazy evaluation) rather than call-by-name evaluation. This would require an additional component of the monad that defines the abstract machine, namely a component to represent a data store. Such an extension would render the model more operational in flavor, incorporating instructions to allocate new storage cells, distinction between storage labels and values, etc. It would be interesting to know how to construct a proof of coherence for such a model that was relative to the proof given here for the simpler model that ignores data stores. In a relative proof, one should only need to prove that instruction sequences of the more detailed model were coherent with those of the

simpler one. A related issue is how to construct a coherence proof for implementation of a programming language with recursive definitions.

A second question concerns abstract machine models for supercombinator reducers. These introduce the complication of multi-argument combinators. An unsaturated combinator application is a normal form, while a saturated application is reducible. The model must account in some way for the number of arguments needed to saturate an application. It would be interesting to know an elegant way to embed this information in a categorical model.

The framework that we have constructed from quasi-closed categories with monads is not the only one possible. An alternative would be to formulate the computational model in a linear category with '!' and '?' modalities. A linear category embeds a fully dual structure. How do these dual modalities relate to a monadal structure? What is the computational significance of the tensor product and sum?

## Acknowledgements

The authors have benefited from conversations with many people during the long course of development of this research. We are particularly indebted to Andrzej Filinski, whose seminal ideas on categorical duality inspired the whole enterprise, and to Eugenio Moggi, whose insights and detailed criticisms were essential to us. Any shortcomings remaining in the paper are ours alone, however.

## References

[1] Asperti, A., *Categorical topics in computer science*, University of Pisa, Dept. of Informatics, Ph.D thesis, 1989.

[2] Barr, M. and Wells, C., *Category Theory for Computing Science*, Prentice-Hall International, 1990.

[3] Cousineau, G., Curien, P. L. and Mauny, M., "The categrical abstract machine," in *Functional Programming Languages and Computer Architecture*, vol. 201, J. Jouannaud (ed.), Springer-Verlag, Nancy, 1985, pp. 50-64.

[4] Cousineau, G., Curien, P. L. and Mauny, M., "The categrical abstract machine," *Science of Computer Programming*, vol. 8, 2 (1987), pp. 173-202.

[5] Curien, P., *Categorical Combinators*, CNRS - Universite Paris VII, Ph.D thesis, 1985.

[6] Fairbairn, J. and Wray, S., "Tim: A simple, lazy abstract machine to execute supercombinators," in *Functional Programming Languages and Computer Architecture*, vol. 274, Springer-Verlag, Portland, Oregon, 1987, pp. 34-45.

[7] Filinski, A., *Declarative Continuations and Categorical Duality*, M.S. thesis, Computer Science Department, University of Copenhagen, 1989.

[8] Griffin, T. G., "A formulae-as-types notion of control," *ACM Symp. on Prin. of Programming Languages*, 1990, pp. 47-58.

[9] Hannan, J. and Miller, D., "From operational semantics to abstract machines: preliminary results," *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, Nice, 1990, pp. 323-332.

[10] Hannan, J., "Making abstract machines less abstract," in *Functional Programming Languages and Computer Architecture*, vol. 523, Springer-Verlag, 1991, pp. 618-635.

[11] Huet, G., *Formal Structures for Computation and Deduction*, INRIA, Rocquencourt, 1986.

[12] Johnsson, T., *The G-machine -- an abstract architecture for graph-reduction*, Dept. of Computer Sciences, Chalmers Univ. of Technology, Gothenburg, 1983.

[13] Jones, S. P., in *The implementation of functional programming languages*, Prentice-Hall International, Hemel Hempstead, 1987.

[14] Lafont, Y., "The linear abstract machine," *Theoretical Computer Science*, vol. 59(1988), pp. 157-180.

[15] Landin, P. J., "The mechanical evaluation of expressions," *Computer Journal*, vol. 6(1964), pp. 308.

[16] Lester, D., *Combinator graph reduction: A congruence and its applications*, Oxford University, PRG, Ph.D thesis, 1988.

[17] MacLane, S., *Categories for the Working Mathematician*, Springer-Verlag, 1971.

[18] Marti-Oliet, N. and Meseguer, J., "Duality in closed and linear categories," SRI-CSL-90-01, SRI International, 1990.

[19] Mauny, M., *Compilation of functional languages in categorical combinators: Application to the language ML*, University of Paris VII, Ph.D thesis, Paris, 1985.

[20] Moggi, E., "Computational lambda-calculus and monads," *LICS'89*, 1989, pp. 14-23.

[21] Moggi, E., "An abstract view of programming languages," LFCS-90-113, Department of Computer Science, University of Edinburgh, 1990.

[22] Moggi, E. and Agapiev, B., *Interpretation of symmetric lambda calculus in some monads*, Oregon Graduate Institute, 1991.

[23] Moggi, E., "Notions of computations and monads," *Information and Computation*, vol. 93, 1 (1991), pp. 55-92.

[24] Stoye, W. R., Clarke, T. J. W. and Norman, A. C., "Some practical methods for rapid combinator reduction," *Proc. 1984 ACM Sympos. on Lisp and Functional Programming*, 1984, pp. 159-166.

[25] Turner, D. and languages, A., *Software - Practice and Experience*, vol. 9(1979), pp. 31-79.

[26] Wadler, P., "Comprehending monads," *Proc. 1990 ACM Sympos. on Lisp and Functional Programming*, 1990.

[27] Warren, D. H. D., "An absract Prolog instruction set," SRI Tech. Note 309, SRI International, 1983.