A Survey of Categorical Computation: Fixed Points, Partiality, Combinators, ... Control?

Dwight Spencer

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 90-017

September, 1990

A Survey of Categorical Computation: Fixed Points, Partiality, Combinators, ... Control?

Dwight Spencer
Oregon Graduate Institute *

August, 1990

Abstract

There has been much recent activity to develop foundational theories and mechanisms of computation based solely on category theory, e.g. the typed lambda calculus becomes derivable in this new computational realm. Unfortunately, the "no-holds-barred" world of the untyped lambda calculus is not also easily subsumed by categorical axiomizations: the computationally attractive closed cartesian category is fraught with the danger of degenerate collapse if we insist upon *both* strong termination properties, e.g. universal fixed points, and flexible data structures, such as infinite lists and direct sums. Several recent attempts to bypass the CCC's limitations either by internal or external means that can mutually support reasonable recursion with reasonable data structures are surveyed in this paper.

1 Introduction

Probably the first major effort to implement a categorical model of computation is the Categorical Abstract Machine by Curien [Cur86]. It is based upon a weak categorical combinatory logic, viz. lacking surjective pairing and extensionality, that arose as a direct semantic-to-syntactic translation of the lambda calculus of tuples. The computational mode was combinator term reduction through rewriting using a direct left-to-right parse algorithm, initially making the evaluation strategy inefficiently eager[1]. Application is therefore simply juxtaposition, losing the full expressiveness of β-reduction that computes via substitution. Its overly strong bias towards the lambda calculus was another factor that limited its expressiveness. On one hand the CAM demanded the existence of categorical products but on the other it had no coproducts for developing many useful data structures. Nevertheless, the high acceptance and efficiency of the CAM-based ML compiler, CAML, gives significant encouragement towards developing a highly-programmable categorical computing paradigm. Some prominent workers in categorical computing now believe "category theory

*Author's address: Department of Computer Science and Engineering, Oregon Graduate Institute, Beaverton, Oregon, 97006-1999. Electronic mail: dwights@cse.ogi.edu

[1] Lazy implementations now exist.

comes, logically, before the λ-calculus" [Mog89a, Mog89b]. This author is thereby motivated to find λ-calculi-independent models of computation in categories expressive enough to build machines capable of a variety of evaluation strategies.

There seem to be several reasonable directions in the search for such computational models. Section 2 points out some baseline categorical constraints we must always keep in mind if we wish to mix convergent computation with rich data structures of sums and products. Sections 3 through 5 present a sample of recent work that passes successively through a spectrum starting from a very "internal" or object-based view and ending with a very "external" or morphism-based approach. This direction very likely corresponds also to increasing implementability. The final section states the thesis that to build a true categorical machine, we must categorically include *control*, or *continuations*, including both its bounded and unbounded forms.

This paper should be accessible to a reader having approximately the categorical experience that might be offered by one of the Pierce or Srinivas tutorials, the well-developed new category theory text by Barr and Wells, and the first three chapters of Barr and Wells' monograph on toposes, triples, and theories [Pie90, BW90, BW85]. It is intended to introduce and guide students to some current research activity in categorical computation and to raise new questions concerning how one might interpret powerful control mechanisms with a categorical model.

2 The Closed-Cartesian Straitjacket

Lambek's [Lam86] demonstration of the exact correspondence of cartesian closed categories with typed lambda calculi brings forth a naive suggestion of cartesian closure as a setting adequate for implementating a general categorical computation paradigm based on beta reduction. The reducing map would be *eval*, representing the morphism component of the universal arrow from the $_ \times A$ functor to the result object B, viz.

$$[A \Rightarrow B] \times A \longrightarrow B.$$

Extending the suggestion, it would seem critical for a CCC to possess a *natural number object* (NNO) to always provide primitive recursion. That is, an NNO N is an object N along with two specified morphisms $0 : 1 \to N$ and $s : N \to N$ having the universal property that for any pair $f : A \to B$ and $g : B \to B$ there exists a unique $h : A \times N \to B$ such that the diagram

$$\begin{array}{ccccc}
A \times 1 & \xrightarrow{id_A \times 0} & A \times N & \xrightarrow{id_A \times s} & A \times N \\
\cong \downarrow & & \downarrow h & & \downarrow h \\
A & \xrightarrow[f]{} & B & \xrightarrow[g]{} & B
\end{array}$$

commutes. This yields the familiar equations $h(x, 0) = f(x)$ and $h(x, n+1) = g(h(x, n))$ where $x : 1 \to A$, n is the composition of n s's with the 0-morphism, and $x + 1$ is the map $h_{x+} \circ s \circ 0$ derived from the universal NNO map h_{x+} for the case $f = x$ and $g = s$.

Yet reasonably expressive computation demands data structures constructed from both products and coproducts. Also, it often demands convergence being expressible as computational fixed points. Unfortunately, the combined presence of useful properties with fixed points within *any* category often forces its collapse to the trivial category $\mathbf{1}$.

An object A is said to have the *fixed point property* if for every morphism $f : A \rightarrow A$ there exists a morphism $Y(f) : 1 \rightarrow A$ such that $f \circ Y(f) = Y(f)$. A category is said to have fixed points if every object has the the fixed point property. In particular, there is a *global element* $\perp_A \equiv Y(id_A) : 1 \rightarrow A$ for each object A. Thus the fixed point operator Y makes the terminal object 1 a weak initial object that potentially can make many objects isomorphic to 1! The major collapse result is given below.

Theorem 2.1 *[HP86] A cartesian closed category with fixed points collapses to the trivial category* $\mathbf{1}$ *if it contains any of the following:*

- *an initial object*
- *a boolean algebra (coproduct) object,* 2 *(*$= 1 + 1$*)*
- *equalizers for any pair of morphisms*
- *a natural number object (NNO)*

Much of the impact of this theorem has been experienced in domain theory where domains and continuous functions are cartesian-closed and all objects have the fixed point property [GMS89]. Not permitting an initial object implies a domain cannot be an empty set, i.e. all domains must be lifted. Not allowing a boolean algebra object prohibits true categorical coproducts in domain categories: the coexistence of coalesced and separated sums - neither a coproduct - in domains are alternative means by which summed structures can be built. If equalizers are not guaranteed, then neither are finite categorical limits, particularly pullbacks. In domains, this is exemplified by the possible occurrence of multiple incoherent minimal sets of information that individually determine the same result value (least fixed point) of a continuous function. Also, unlike an NNO, a recursive definition of continuous function over domains does not always uniquely define a function.

By inspecting Huwig and Poigne's proofs of Theorem 2.1, we can isolate the key condition that will collapse *any* category having fixed points:

Theorem 2.2 *If any category with fixed points has a natural isomorphism* $2 \times A \cong A + A$ *for every object* A*, then the category collapses to the trivial category* $\mathbf{1}$.

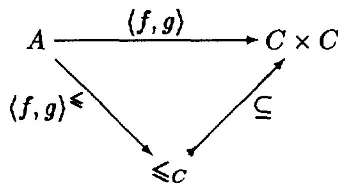
The rest of this paper mainly discusses ways in which the CCC "straitjacket" might be loosened to provide meaningful computation.

3 Computing Fixed Points

Having the fixed point property for every object, or more precisely every endomorphism, in a CCC is by itself an unnecessarily rigid model for computation. Often we want the convergence of only well-behaved morphisms — such as continuous ones — between ordered objects. A well-known approach

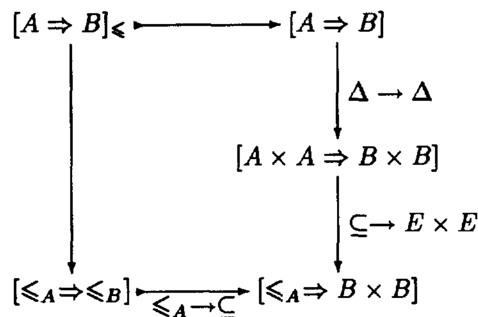
to directly characterize terminating computation using "complete partial orders" and "continuity" in CCCs having finite limits and a natural numbers object has been recently consolidated by Barr [Bar90]. This section sketches the key concepts and results of that work.

The definition of a partially ordered object in **C**, a CCC possessing the above-stated properties, follows closely the use of relations to define orders in domain theory. A *partial order on an object* C in **C** is a subobject $\leqslant_C$ of $C \times C$, i.e. a monic $\subseteq$ from $\leqslant_C$ to $C \times C$ exists, that induces a partial order for any homset $Hom(A, C)$ in the following way: if $f, g : A \rightarrow C$, then $f \leqslant g$ in $Hom(A, C)$ if and only if $\langle f, g\rangle$ factors through $\leqslant_C$ (see the diagram below). Since the *elements* of C are the morphisms going to C, then the partial order definition establishes a partial order for the set of elements of C. Thus to say for morphisms f and g that $f \leqslant g$ requires that they have common domains and codomains, and their mutual codomain is a partially ordered object.



With this definition, we will take full advantage of exponentials to build objects that internalize order-preserving maps and l.u.b.-preserving (continuous) maps. To capture the order-preservation of a morphism $f : A \rightarrow B$, we need to express the statement "$x \leqslant y$ implies $f(x) \leqslant f(y)$" categorically by somehow internally factoring $f \times f : A \times A \rightarrow B \times B$ through the object $[\leqslant_A \Rightarrow \leqslant_B]$. We therefore automatically assume that A and B are partially ordered objects for which special factorizing subobjects $\leqslant_A$ and $\leqslant_B$ exist.

The object that contains f and all other order-preserving morphisms from A to B, designated as $[A \Rightarrow B]_{\leqslant}$, can be built as a pullback:



The bottom and lower right morphisms are the ones induced from the canonical morphisms between the corresponding hom-sets via the product - exponential adjunction and Yoneda's Lemma. For example,

$$
\begin{array}{ccc}
Hom(\leqslant_A, \leqslant_B) & \xrightarrow{Hom(\leqslant_A, \subseteq)} & Hom(\leqslant_A, B \times B) \\
\cong \uparrow & & \downarrow \cong \\
Hom(1, [\leqslant_A \Rightarrow \leqslant_B]) & \longrightarrow & Hom(1, [\leqslant_A \Rightarrow B \times B])
\end{array}
$$

$$
\Big\downarrow \text{Yoneda embedding}
$$

$$
[\leqslant_A \Rightarrow \leqslant_B] \xrightarrow{\leqslant_A \rightarrow \subseteq} [\leqslant_A \Rightarrow B \times B]
$$

The upper right morphism ($\Delta \rightarrow \Delta$) of the pullback internalizes the map that diagonalizes $f : A \rightarrow B$ to $f \times f : A \times A \rightarrow B \times B$. That map is the exponential transpose of the following composition:

$$
\begin{array}{c}
[A \Rightarrow B] \times A \times A \\
\Big\downarrow \Delta \times id_{A \times A} \\
[A \Rightarrow B] \times [A \Rightarrow B] \times A \times A \\
\Big\downarrow \langle \pi_1, \pi_3, \pi_2, \pi_4 \rangle \\
[A \Rightarrow B] \times A \times [A \Rightarrow B] \times A \\
\Big\downarrow eval \times eval \\
B \times B
\end{array}
$$

The definition pullback can now be directly interpreted. If f is an element of $[A \Rightarrow B]$, then it is also an element of $[A \Rightarrow B]_{\leqslant}$ if and only if $f \times f$, when restricted to the order relation $\leqslant_A$, maps into the order relation on B, i.e. f preserves order. Note that this pleasant situation depends significantly on the fact that pullbacks preserve monics, and so by the definition $[A \Rightarrow B]_{\leqslant}$ is a subobject of $[A \Rightarrow B]$.

Along with a partial ordering for an object A we define a ***bottom*** for A. It is a map $\perp : 1 \rightarrow A$ that possesses the following leastness property: for any object B and any morphism $f : B \rightarrow A$, $\perp \circ \Diamond_B \leqslant f$, where $\Diamond_B$ denotes the unique terminal map $B \rightarrow 1$. An intuition for the bottom condition can be reached by loosely interpreting the $\Diamond_B$ as a "$\lambda\, x : B.$" binding operator, so that $\perp \circ \Diamond_B$ can be thought of as the bottom constant function on B.

We now consider how to form a partial ordering for a particular object, the natural number object N. It is well-known that a commutative associative map $+ : N \times N \rightarrow N$ is derivable from the NNO's universal property which satisfies the laws of primitive recursive arithmetic. Derivations of $+$, *pred*, $\dot{-}$ (monus), the basic arithmetic identities and algebraic cancellation properties were

performed by Lambek [Lam86] for a CCC with an NNO and improved by Roman [Rom89] for the weaker case of categories with an NNO having only finite products.

A suitable natural number ordering is the subobject $p : N \times N \rightarrowtail N \times N$ where $p \triangleq (id_N, +)$. That is, if $n, m : A \to N$ are two elements of N (intuitively, the natural numbers) then $(n, m) \overset{p}{\mapsto} (n, n + m)$. The partial ordering property can be seen in the diagram below for a pair of "numbers" n and m':

$$\begin{array}{ccc} A \times A & \xrightarrow{\langle n, m' \rangle} & N \times N \\ & {\scriptstyle n \times m} \searrow \quad \nearrow {\scriptstyle p = (id, +)} & \\ & N \times N & \end{array}$$

Clearly the map $\langle n, m' \rangle$ factors through the partial order object of N (here $N \times N$ itself) if and only if $m' = n + m$, i.e. exactly whenever $n \leqslant m'$. We denote the natural numbers object N as the partial ordered object ω and the object $[\omega \Rightarrow A]_{\leqslant}$ of *increasing sequences* of A by A^ω. Under the abbreviated notation we can say that A^ω is a subobject of A^N, the object of *sequences* of A.

Barr proves the following basic and hoped-for result solely from this fundamental definition of partial ordered object:

Theorem 3.1 *Let D be a partially ordered object in a CCC with finite limits and an NNO. Then if $f : N \to D$ is a morphism such that $f(n) \leqslant f(sn)$ for every element $n : A \to N$ of N. Then f is order-preserving*[2].

Corollary 3.1 *With the same general hypotheses above, let $f, g : N \to D$ have the properties that $f(0) \leqslant g(0)$ and $f(n) \leqslant g(n)$ implies $f(sn) \leqslant g(sn)$, then $f \leqslant g$.*

We finally come to the categorical definition of an ω-CPO. D is said to be an ω-complete partial order if there exists a (l.u.b.) map $\bigvee : D^\omega \to D$ that satisfies two properties: (1) for any arrow $f : A \to D^\omega$ with its exponential transpose $\tilde{f} : A \times \omega \to D$, we have $\tilde{f} \leqslant \bigvee \circ f \circ \pi_1$ and (2) if $g : A \to D$ is any morphism such that $\tilde{f} \leqslant g \circ \pi_1$, then $\bigvee \circ f \leqslant g$. The composition of (1) is precisely

$$A \times \omega \xrightarrow{\pi_1} A \xrightarrow{f} D^\omega \xrightarrow{\bigvee} D$$

To reassure ourselves that this definition corresponds exactly to the conventional least-upper-bound conditions, consider each of the two properties. The first one can be written as

$$\tilde{f}(\langle a, n \rangle) \leqslant \bigvee(f(a))$$

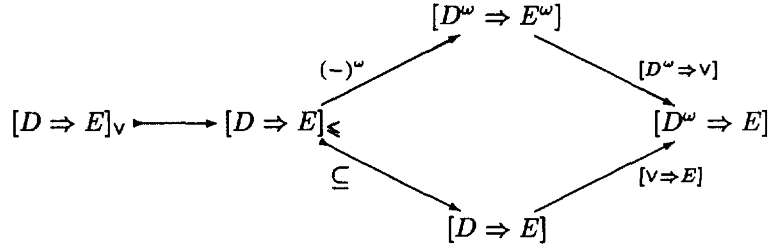
and read as "any member of the increasing sequence is $\leqslant$ the l.u.b. of the sequence". The second one can be expressed as

[2] We write "$f(n)$" because the Yoneda embedding implies $f(n) = f \circ n$.

$$\tilde{f}(\langle a, n\rangle) \leqslant g(a) \Rightarrow \bigvee(f(a)) \leqslant g(a)$$

and read as "if every member of the increasing sequence is $\leqslant$ the value of g at a, then the l.u.b. of the sequence is $\leqslant$ the value of g at a."

We would now expect a definition of a continuous morphism f between ω-CPOs to express the commutativity of f with the l.u.b. operator $\bigvee$. This is exactly what is expressed in the diagram below that defines the object of *ω-continuous* morphisms from a ω-CPO D to an ω-CPO E:



This object, $[D \Rightarrow E]_{\vee}$, is defined as the equalizer of the two compositions forming the diamond at the right. Since equalizers are monics, $[D \Rightarrow E]_{\vee}$ is a subobject of $[D \Rightarrow E]_{\leqslant}$. The interpretation of the diagram is straightforward. The map $(-)^{\omega}$ is induced by taking an order-preserving morphism f from $[D \Rightarrow E]_{\leqslant}$ and using it to map increasing sequences on D, for example $\{d_i\}$, to increasing sequences on E, here $\{f(d_i)\}$. The $[D^{\omega} \Rightarrow \vee]$ morphism post-composes the sequences-to-sequences map with the l.u.b. operation. Thus the result of the top of the diamond is a morphism in $[D^{\omega} \Rightarrow E]$ that maps an increasing sequence into the l.u.b. of its f-image. The bottom of the diamond simply pre-composes f with the l.u.b. operation, i.e. the result is a morphism that first "l.u.b's" an increasing sequence in D and then maps the bound via f. Finally, if f were to be chosen from the equalizer, $[D \Rightarrow E]_{\vee}$, of the top and bottom of the diamond, then the result morphisms are the same: f commutes with $\bigvee$. By the universal property of the equalizer, this subobject of the order-preserving morphisms from D to E represents precisely these ω-continuous ones.

We can now inspect Barr's main result: the existence of a least fixed point operator for ω-continuous endomorphisms on ω-CPOs:

Theorem 3.2 *Suppose D is an ω-CPO with a bottom element $\perp : 1 \rightarrow D$. Then there exists a morphism $fix : [D \Rightarrow D]_{\vee}$ such that (1) $f(fix(f)) = fix(f)$ and (2) if $f(d) = d$, then $fix(f) \leqslant d$.*

The proof requires several distracting technical lemmas, but a brief hint of the argument is appropriate since it shows in a categorical setting the conventional construction of a least fixed point. Let $f : A \rightarrow [D \Rightarrow D]_{\vee}$ be an element of $[D \Rightarrow D]_{\vee}$. The product-exponential adjunction permits this element to correspond to its ω-continuous - and consequently order-preserving - transpose $\tilde{f} : A \times D \rightarrow D$ where A is provided with the discrete ordering and $A \times D$ with the resultant product ordering.

We now construct a morphism $\tilde{g} : A \times N \rightarrow A$ that imitates internally the effect of successive applications of f to $\perp$. We use the NNO to define g according to the following recursive specification:

$$\begin{aligned}\tilde{g}(a,0) &= \bot \\ \tilde{g}(a,s(n)) &= \tilde{f}(a,\tilde{g}(a,n))\end{aligned}$$

Computing the first few values of g shows us that the recursive computation at the transpose level corresponds to the iterations of f against $\bot$ at the exponential level:

$$\begin{aligned}\tilde{g}(a,0) &= \bot \\ \tilde{g}(a,s(0)) &= \tilde{f}(a,\bot) \longleftrightarrow f(\bot) \\ \tilde{g}(a,s(s(0))) &= \tilde{f}(a,\tilde{f}(a,\bot)) \longleftrightarrow f(f(\bot))\end{aligned}$$

The universal property of the NNO forces the uniqueness of the morphism h below. By the commutativity of this diagram, the second component of h is seen to be the desired map $\tilde{g}$.

$$\begin{array}{ccccc} A\times 1 & \xrightarrow{id_A\times 0} & A\times N & \xrightarrow{id_A\times s} & A\times N \\ \cong\downarrow & & \downarrow h=(id_A,\tilde{g}) & & \downarrow h=(id_A,\tilde{g}) \\ A & \xrightarrow[\langle id_A,\bot\circ\Diamond_A\rangle]{} & A\times D & \xrightarrow[\langle \pi_1,\tilde{f}\rangle]{} & A\times D \end{array}$$

The proof proceeds to show that because $\bot\circ\Diamond_A$ is the least element of D based on A, then $\tilde{g}$ is order-preserving. This is equivalent to saying the curry of $\tilde{g}$, $g:A\rightarrow D^N$ factors through D^ω. The fixed point of f is then definable as $\bigvee\circ g:A\rightarrow D$. That is, we are simply computing the l.u.b. of the sequence $\bot$, $f(\bot)$, $f(f(\bot))$, ... ! The proof concludes by verifying the least fixed point properties of $\bigvee\circ g$.

4 Categories of Partial Maps

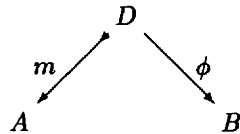
Recursive programming is intrinisically a partial computation: the *domain of definition* is often only a sub-domain of a computation's domain. Yet the classical [Mac71] categorical view of a function is unrealistically a *total* function. This narrow philosophy has motivated many authors to build categories from a *partial morphism* foundation. The loss of totality precludes general cartesian products[3], so these categories are accompanied by some "near" or tensor product to provide real data structure. This idea contrasts with the preceding section where computation is extracted from a CCC by building within it special objects that internally capture the well-behaved computable functions, i.e. a recursion theory that "uses elements". On the other hand, the partial map categories encourage "element-free" approaches to computation.

Various significant constructions of categories of partial maps include the *dominical categories* of DiPaola and Heller [DH87], the *concrete categories of domains* of Moggi [Mog86, Mog88b], the *partial cartesian closed categories* of Asperti, Longo, and Moggi [LM84, AL86], and the *partial*

[3] Uniqueness properties disappear.

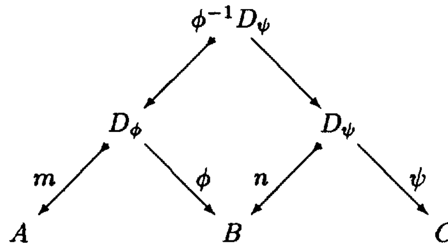
cartesian categories of Curien and Obtulowicz [CO89]. These constructions are complex and not easy to express. This section discusses the work of Robinson and Rosolini [Ros86, RR88] that unifies these above-mentioned categories (and others) and the general category of partial maps with a fundamental axiomization: the *p-category* [4].

First we will explain the basic category of partial maps **Ptl(A)** over a category **A** from which all special partial morphism theories are motivated. The objects of **Ptl(A)** are those of **A**. A *partial map* $[m,\phi]: A \rightharpoonup B$ between the objects A and B of **A** is an equivalence class of pairs



where all equivalent pairs are related by isomorphisms between their respective *domain objects* (shown here as D) such that the obvious triangles commute. Note that the domain of definition is represented as a subobject of the partial morphism's domain.

The composition of two partial maps $[m,\phi]: A \rightharpoonup B$ and $[n,\psi]: B \rightharpoonup C$ is completed by pulling back n along ϕ to achieve the morphism pair representing a composite partial morphism[5]:



It is immediate that the composition is well-defined under the equivalence and that the identity map for each object A is $[id_A, id_A]$. It should also be clear that the original category **A** embeds into **Ptl(A)** where $A \mapsto A$ and $f: A \to B \mapsto [id_A, f]: A \rightharpoonup B$. Its image is called the *total maps* of **Ptl(A)**.

However, the generality problem of the category of partial maps is evident: it's too big! We often wish to restrict our focus to classes of partial morphisms (continuous, monotonic, etc.) of particular interest that are defined on an appropriate set of domains (open sets, natural number object, etc.). Thus we form a subcategory of **Ptl(A)** with a selected *admissible* class $\mathcal{M}$ of subobjects that — as we can now easily infer from the definition of composition above — must be closed under identities[6], composition in **A**, and pullbacks. In the particular case that **A** has cartesian products, the admissible class is called a *dominion*. Therefore the new category, $\mathcal{M}$-**Ptl(A)**, are those partial maps whose domains of definition lie in $\mathcal{M}$.

[4] The invention of the p-category first appeared in Rosolini's thesis.

[5] The category **A** must possess pullbacks, of course.

[6] That is, $\mathcal{M}$ must include all the identity maps of **A**. This permits **A** to exist as an embedded category within this new subcategory.

Another important assumption is the presence of a categorical product (i.e. cartesian) on the generating category **A**. It provides an algebraic richness sufficient for axiomization without creating much loss in generality. In fact, it is straightforward to show that $\mathcal{M}$ is also closed under the assumed product of **A** and that any category of partial maps can be fully embedded in a category of partial maps over a category with categorical products[7].

Now consider the extension of the A-product to all of $\mathcal{M}$-**Ptl(A)** by defining for pairs of partial maps $[m, \phi] \times [n, \psi] \triangleq [m \times n, \phi \times \psi]$. The extended product bifunctor is no longer necessarily cartesian. Robinson and Rosolini point out that the crux is that the projections are no longer natural[8] in *both* of its arguments. Suppose that the map ϕ is truely partial. Because the cartesian product of **A** induces the naturality of π_1 within $\mathcal{M}$-**Ptl(A)** in its first argument, we would have in $\mathcal{M}$-**Ptl(A)** the truely partial composition $\pi_1(id \times \phi) = id \circ \pi_1 = \pi_1$, contradicting the totality of π_1 (embedded as $[id, \pi_1]$). Fortunately the diagonal map $\Delta : (-) \rightarrow (-) \times (-)$, the associativity map $\alpha = ((id \times \pi_1) \times \pi_2\pi_2)\Delta : X \times (Y \times Z) \rightarrow (X \times Y) \times Z$, and the commutativity map $\tau = (\pi_2 \times \pi_1)\Delta : X \times Y \rightarrow Y \times X$ remain natural in all variables under the extension. With all these observations, we are motivated to present the following definition:

Definition 4.1 *A* p-*category is a category* **C** *having a near-product bifunctor* $\times : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}$, *a natural diagonal map* Δ, *two families of projections* $\{\pi_{1_Y} : (- \times Y) \rightarrow (-) \mid Y \in Obj(\mathbf{C})\}$ *and* $\{\pi_{2_X} : (X \times -) \rightarrow (-) \mid X \in Obj(\mathbf{C})\}$ *natural only in the indicated arguments, and the associativity and commutativity maps* α *and* τ *(defined above) as isomorphisms natural in all arguments, where*

$$\begin{array}{ll} \pi_1\Delta = id = \pi_2\Delta & (\pi_1 \times \pi_2)\Delta = id \\ \pi_1(id \times \pi_1) = \pi_1 & \pi_1(id \times \pi_2) = \pi_1 \\ \pi_2(\pi_1 \times id) = \pi_2 & \pi_2(\pi_2 \times id) = \pi_2 \end{array}$$

As we would demand from this definition, the category of partial maps, $\mathcal{M}$-**Ptl(A)**, is indeed a *p*-category induced by the canonical embedding of **A** and the extension of **A**'s cartesian product, as described earlier. We should also note that being a *p*-category indicates the existence of an *auxiliary product structure* on the category that may or may not coincide with any native product structure of the category.

But to complete the abstraction of partiality, we must now decide what are the *total maps* of a *p*-category! With our example of $\mathcal{M}$-**Ptl(A)**, they are the maps of **A** embedded as $[id, \phi]$. The pathological example of $\pi_1(id \times \phi)$ for a non-total map ϕ suggested to Di Paola and Heller a new way to describe the domain of ϕ.

Definition 4.2 *For a map* $\phi : A \rightarrow B$ *in a* p-*category* **C**, *the domain* $dom(\phi) : A \rightarrow A$ *of* ϕ *is the composition* $\pi_1(id \times \phi)\Delta$.

Let us first apply this definition to our *p*-category example $\mathcal{M}$-**Ptl(A)**. The computation of $dom[m, \phi]$ is shown below where the outermost oblique composition is $dom([id, \phi])$, accomplished by the three pullback squares. The outermost vertical composition is its first component (domain of

[7] Simply apply the Yoneda embedding of **A** into the topos $Func(\mathbf{A}^{op}, Set)$ where each object A maps to the functor $Hom(-, A)$. From its definition, a topos has cartesian products.

[8] A functional programmer might say "no longer independent of evaluation strategy".

definition), and the uppermost horizontal composition is its second component. The parenthetical legends identify the corresponding morphisms as morphisms in $\mathcal{M}$-**Ptl(A)**. By using the properties of the cartesian product in **A** we quickly see $dom([id, \phi]) = [m, m]$.[9] Thus our partial map $[m, \phi]$ is total, i.e. $m = id$, if and only if $dom([m, \phi]) = [id, id] = id$. This example motivates classifying a map ϕ of a p-category as *total* whenever $dom(\phi) = id$.

$$
\begin{array}{ccccccccc}
D_\phi & \xrightarrow{\langle m, id\rangle} & A \times D_\phi & \xrightarrow{id \times \phi} & A \times B & \xrightarrow{\pi_1} & A & & \\
\big\downarrow{\scriptstyle id} & & \big\downarrow{\scriptstyle id} & & \big\downarrow{\scriptstyle id} & & [id, \pi_1] & & (\pi_1) \\
D_\phi & \xrightarrow{\langle m, id\rangle} & A \times D_\phi & \xrightarrow{id \times \phi} & A \times B & & & & \\
\big\downarrow{\scriptstyle m} & & \big\downarrow{\scriptstyle id \times m} & & [id \times m, id \times \phi] & & (id \times \phi) & & \\
A & \xrightarrow{\Delta} & A \times A & & & & & & \\
\big\downarrow{\scriptstyle id} & & [id, \Delta] & & (\Delta) & & & & \\
A & & & & & & & &
\end{array}
$$

The near product of the p-category is sufficiently powerful to provide the domain operator with many easy-to-prove properties that imitate very closely what a domain of definition would be expected to have, say, in the familiar p-category **Ptl(Sets)**.

Theorem 4.1 *The following properties hold for a domain operator of a* p*-category:*

1. $\pi_1(id \times \phi) = \pi_1(id \times dom(\phi))$
2. $(id \times dom(\phi))\Delta = \Delta\ dom(\phi)$
3. $dom(id) = id$, $dom(\pi_1) = id$, $dom(\pi_2) = id$, $dom(\Delta) = id$
4. $dom(\phi\psi) = dom((dom(\phi)\psi)$
5. $(dom(\phi))\psi = \psi\ dom(\phi\psi)$
6. $dom(\phi \times \psi) = dom(\phi) \times dom(\psi)$
7. $dom(\phi)\ dom(\psi) = dom(\psi)\ dom(\phi) = dom(dom(\phi)\ dom(\psi))$
8. $dom(\phi)\ dom(\phi) = dom(\phi)$
9. $dom(dom(\phi)) = dom(\phi)$
10. $\phi\ dom(\phi) = \phi$

[9] Technically, the equality is between equivalence classes.

11. $dom(\phi\psi) = dom(\psi)\ dom(\phi\psi)$

12. $\phi = dom(\psi)$ *if and only if* $\phi = dom(\phi)$

Robinson and Rosolini interpret property (1) as showing the amount by which naturality of the second variable of π_1 fails. Property (2) serves as an indication that domains can have properties that are typically satisfiable only by identity maps, i.e. domains behave like identities. Property (3) show the totality of the identity, the projections, and the diagonal. The remaining properties can be intuitively verified from our past experiences with partial functions, especially property (10) which states a partial function operating only on its domain of definition is simply the same function.

Now that we have an algebraic notion of partial function, it is proper to ask how well does it fit with the classic notion based on categories of the form $\mathcal{M}$-**Ptl(A)**. We saw earlier that $\mathcal{M}$-**Ptl(A)** is a p-category if **A** is cartesian. However, does *any* p-category behave as a category of the form $\mathcal{M}$-**Ptl(A)** where **A** has merely a product bifunctor, possibly non-cartesian? Rephrased, the question becomes: is there a direct correspondence of the properties of p-categories with the properties of the categories of partial maps relative to admissible classes of domains? The answer by Robinson and Rosolini is mostly positive. Starting with a p-category **C**, they constructed a category with a weak product and an admissible class to form a partial category into which **C** could be embedded. A sketch of that construction is described in the following paragraphs.

The *category of domains*, **Dom(C)**, that will generate the desired partial map category has as objects the maps of **C** of the form $dom(\phi): X \rightarrow X$. Such an object is said to be a *domain on* X. A map $\phi: dom(\beta) \rightarrow dom(\gamma)$ in **Dom(C)** is a map $\phi: X \rightarrow Y$ in **C** where $dom(\beta) = dom(\phi)$ and $\phi = (dom(\gamma))\phi$. This is an algebraic way of saying ϕ is defined on $dom(\beta)$ and produces values in $dom(\gamma)$. The composition in **Dom(C)** is defined as the composition in **C**. By the definition of morphism for **Dom(C)** and Property (10), the identity on $dom(\beta)$ is $dom(\beta)$ itself, giving us a category in which some of objects and morphisms coincide! Finally, Property (6) shows that **Dom(C)** has a product.

In preparation for finding an admissible class of subobjects in **Dom(C)**, we now follow the hint offered by the identity maps and characterize *those morphisms that coincide with the objects* in **Dom(C)**. First, back in **C**, we define the *extension order* on maps from X to Y by

$$\beta \leqslant \gamma \Longleftrightarrow \gamma\ dom(\beta).$$

Expanding the order definition and using π_1's naturality in its first argument we can equivalently state

$$\beta \leqslant \gamma \Longleftrightarrow \pi_1(\gamma \times \beta)\Delta = \beta.$$

This immediately yields the intuitively satisfying fact that β is a domain, i.e. $\beta = dom(\beta)$ by Property (12), if and only if $\beta \leqslant id$. Consequently we always have $dom(\beta) \leqslant id$. For another example of the consistency of the extension ordering with our set-theoretic intuition, it is easy to show that if $\phi\ dom(\psi) = \theta$ and $dom(\theta) \leqslant dom(\psi)$, then $\theta \leqslant \phi$. This says that if ϕ behaves as θ on the domain of ψ and ψ is defined whenever θ is, then ϕ must be an extension of θ.

We can now state the characterization of morphisms-as-objects in $\mathbf{Dom(C)}$. Its proof is straightforward from the properties of domains listed earlier.

Theorem 4.2 *Let β, γ, and δ be domains on the object X. Then $\delta : dom(\beta) \to dom(\gamma)$ is a morphism in $\mathbf{Dom(C)}$ if and only if $\delta = \beta$ and $\beta \leqslant \gamma$. When $\beta \leqslant \gamma$, then the morphism $dom(\beta)$: $dom(\beta) \to dom(\gamma)$ is a monomorphism in $\mathbf{Dom(C)}$.*

Since we are seeking an admissible class of subobjects, we consider $\mathcal{D}_{\mathbf{C}}$, the collection of monics-as-objects in $\mathbf{Dom(C)}$. Consequently, these monics have the form $\beta = dom(\beta) : dom(\beta) \rightarrowtail dom(\gamma)$. The verification of $\mathcal{D}_{\mathbf{C}}$'s admissibility is direct, but the reader should note while checking that the pullback of β along any morphism $\phi : dom(\delta) \to dom(\gamma)$ in $\mathbf{Dom(C)}$ also lies in $\mathcal{D}_{\mathbf{C}}$ that we do not confuse composition in $\mathbf{C}$ with that in $\mathbf{Dom(C)}$. The resultant pullback below is annotated with names of *inducing* morphisms of $\mathbf{C}$, thereby allowing the indicated composition $\beta\phi$ to be sensible:

$$\begin{array}{ccc} dom(dom(\beta)\phi) & \xrightarrow{\beta\phi} & dom(\beta) \\ {\scriptstyle dom(\beta\phi)} \downarrow & & \downarrow {\scriptstyle \beta} \\ dom(\delta) & \xrightarrow[\phi]{} & dom(\gamma) \end{array}$$

With this in hand, the representation theorem for p-categories can be stated:

Theorem 4.3 *Any* p*-category* $\mathbf{C}$ *can be fully embedded into* $\mathcal{D}_{\mathbf{C}}$*-*$\mathbf{Ptl(Dom(C))}$ *in a p-structure preserving way.*

Such an embedding $Emb : \mathbf{C} \to \mathcal{D}_{\mathbf{C}}-\mathbf{Ptl(Dom(C))}$ is defined as $X \mapsto dom(id_X)$ and $\phi : X \to Y \mapsto [dom(\phi), \phi] : dom(id_X) \to dom(id_Y)$, where the component maps are $dom(\phi) : dom(\phi) \rightarrowtail dom(id_X)$ and $\phi : dom(\phi) \to dom(id_Y)$. Note that $dom(\phi)$ is clearly in $\mathcal{D}_{\mathbf{C}}$ because of Theorem 4.2 and $dom(\phi) \leqslant id$.

So we have a correspondence of the properties of p-categories to those of categories of partial maps. To go the opposite direction, we need to determine what particular circumstances cause a p-category to be exactly of the form $\mathcal{M}$-$\mathbf{Ptl(A)}$. Robinson and Rosolini exploit a definition and theorem of Freyd to establish the opposite correspondence.

Definition 4.3 *[Fre74, FS90] Let $\mathcal{I}$ be a class of idempotent morphisms in a category $\mathbf{C}$. Define the category $\mathbf{Split(C,}\mathcal{I}\mathbf{)}$ where the objects are the elements of $\mathcal{I} \cup \{id_X \mid X \in Obj(\mathbf{C})\}$ and $f : e \to d$ is a morphism if f is a map in $\mathbf{C}$ such that $d\,fe = f$.*

Observe that the objects of $\mathcal{D}_{\mathbf{C}}$-$\mathbf{Ptl(Dom(C))}$ are the domains of $\mathbf{C}$, including the identities, which are all idempotent by property (8). By the definition of $\mathbf{Dom(C)}$ and $\mathcal{D}$, every morphism is of the form $[dom(\phi), \phi] : dom(\beta) \to dom(\gamma)$. First note that this form is completely and uniquely

determined by ϕ, so that $[dom(\phi), \phi]$ can be identified with ϕ. Second, $[dom(\phi), \phi]$ is a map in $\mathcal{D}_{\mathbf{C}}$-**Ptl(Dom(C))** if and only if $\phi \leqslant \beta$ (by Theorem 4.2) and $dom(\gamma)\phi = \phi$. The properties of domains and the definition of $\leqslant$ easily show this equates to the condition that $(dom(\gamma))\phi(dom(\beta)) = \phi$. That is, $\mathcal{D}_{\mathbf{C}}$-**Ptl(Dom(C))** is actually **Split(C,**Obj(**Dom C**)**)**.

We now make use of Freyd's result.

Theorem 4.4 *[Fre74, FS90] Let* **C** *be a category containing a class* $\mathcal{I}$ *of splitting idempotents*[10]. *Then* **C** *is equivalent to* **Split(C,**$\mathcal{I}$**)**.

Hence we have immediately a second representation theorem that states properties of partial maps correspond exactly to those properties of *p*-categories that are independent of domains splitting.

Theorem 4.5 *A* p*-category* **C** *is equivalent to* $\mathcal{D}_{\mathbf{C}}$-**Ptl(Dom(C))** *if and only if all domains of* **C** *split.*

To end this section, it is relevant to briefly refer the reader to the recent work of C. Barry Jay [Jay90] who takes a different point of view of characterizing categories of partial maps. Instead of building auxiliary structures, such as the *p*-category structure, Jay takes advantage of the intrinsic ordering of a partial map category's hom-sets induced by the ordering of domains of definition to define *lax*[11] versions of product, adjunction, functor, and natural transformation from which the structure of categories of partial maps can be fully and uniformly described.

5 Combinator Reduction

We move our attention to "real" computation within a category, or what has been generally termed *categorical programming*. Two major foci of research have been data structures and combinator reduction. This section will restrict its discussion to the very recent work in the latter area by Cockett and Chen on developing a methodology for directly specifying and building abstract machines that compute with categorical combinators to solve specific problems [CC90].

The Cockett/Chen viewpoint has been influenced heavily by the experiences of Hagino in categorical data structures [Hag87]. Hagino elegantly developed a categorical programming language entirely from the minimal foundation of mixed-variance multi-arity functors and adjunctions. He was able to produce a reasonable but limited variety of useful data structures. Cockett and Chen adapted Hagino's chain-reduction methods for their own reduction algorithms and extended significantly the breadth of data structures possible to be specified. A particular improvement is that the new methodology can be applied to computing in distributive and list-arithmetic settings, two data structure environments important to programmers.

They have been motivated from several directions. Inspiration and encouragement came from (1) noting the implementations of Hagino-like data structures in the polymorphic λ-calculus by Wraith

[10] The splitting of an idempotent is equivalent to its being a retraction.

[11] Laxity is the weakest sense of commutativity. Instead of equality (strict commutativity) or isomorphism (pseudo commutativity), we merely require the existence of a morphism.

[Wra89], (2) observing the highly intuitive aspects of programming with the equational logics of categories to specify computational environments for combinator systems, and (3) discovering the flexibility in selecting strongly-normalizing categorical settings to solve particular computational problems. We will outline their methodology for both constructing combinator systems (called *theories*) from the equations of categories and computing with them.

First, a *combinator theory* has three parts: a *type stack*, a *combinator base*, and a set of axioms. The type stack[12] serves as a graded term algebra for expressing the type signatures of combinators (special morphisms) and higher categorical constructs such as products and exponentiation. It is "stacked", or nested, because we wish to take advantage of the categorical hierarchy: the bottom level $\mathcal{T}_0$ of the stack should be thought of as the objects of a category, the second and next higher level $\mathcal{T}_1$ as categories, the third level $\mathcal{T}_2$ as 2-categories, and so on. The terms in $\mathcal{T}_i$ are generated by a finite collection of function symbols $\mathcal{F}_i$ and a countable collection of variables $\mathcal{X}_i$. One may loosely relate the function symbols to type constructors. Within the stack a *type assignment* associates each stack level with the next higher level, making the type stack a directed graph of typing assignments. Also defined for the type stack is provision for multi-level substitution to effect type instantiation that permits a Milner-style algorithm to find the most general unifier of two type terms. The type stack thus becomes the authors' method of avoiding the awkward complexity of the functor calculus used to build type signatures for Hagino's categorical data structures. Here the type stack becomes the type structure for the combinators.

A couple of examples should illuminate the definition of type stack. First we see a prospective type stack for a combinator theory that is based on a cartesian closed category:

$$\begin{array}{rcl} \mathcal{F}_0 & = & \{1, \times, \Rightarrow\} \\ 1 & \mapsto & ([\,], \mathbf{X}) \\ \times & \mapsto & ([\mathbf{X}, \mathbf{X}], \mathbf{X}) \\ \Rightarrow & \mapsto & ([\mathbf{X}, \mathbf{X}], \mathbf{X}) \end{array}$$

A list of three function symbols is presented along with the use of a single type variable of level $\mathcal{T}_1$ that will be eventually assigned a category from which to draw objects[13]. The assignment of the type signatures to the function symbols is shown as well. For example, the product type constructor accepts a pair of two objects (in the authors' notation, a list of two category variables), both from the *same* category because the common variable $\mathbf{X}$ is used. The function 1 has arity[14] 0 and can therefore be identified with a particular constant object. Some type expressions constructible from objects, say X, Y, and Z, from this stack include:

$$(X \Rightarrow Y) \Rightarrow Z \qquad X \times Y \qquad 1 \Rightarrow Z$$

The other type stack example below is more interesting because it demonstrates more of the categorical nesting facility and the flexibility in narrowing the stack to a specific class of problems. This stack could be used by a combinator theory for an adjunction where the functor F is specified

[12] Cockett and Chen's definition is extensive. Only an informal explanation is appropriate here.

[13] Typically the variable assignment arises from a type unification phase during "compilation" of the type stack.

[14] Arity is being abused here. We refer only to parameter-arity, not input-arity.

below to map objects from a category assigned to the variable $\mathbf{C}$ to objects in a category assigned to the variable $\mathbf{D}$. Clearly the functor G travels in the opposite direction to F.

$$\begin{array}{rcl} \mathcal{F}_0 & = & \{F, G\} \\ \mathcal{F}_1 & = & \{\mathbf{F}, \mathbf{G}\} \\ \mathbf{C} & \mapsto & ([\,], \hat{\mathbf{X}}) \\ \mathbf{D} & \mapsto & ([\,], \hat{\mathbf{X}}) \\ F & \mapsto & ([\mathbf{C}], \mathbf{D}) \\ G & \mapsto & ([\mathbf{D}], \mathbf{C}) \end{array}$$

Note that both category variables, $\mathbf{C}$ and $\mathbf{D}$, are type-assigned as constants with the same 2-category via $\hat{\mathbf{X}}$. As would be sensible, our functors should be between categories belonging to the same class of categories. Here are some correct type expressions:

$$G(F(X)) \qquad G(F(G(F(G(F(X)))))) \qquad \mathbf{C}$$

The second major component of a combinator theory, the combinator base, is defined to be a set of combinators with signatures, expressed in type terms from $\mathcal{T}_0$, of the form $([A_1 \to B_1, ..., A_n \to B_n], C \to D)$ for $n \geq 0$. The type-pair constructor "$\to$" is technically a binary function symbol implicitly assumed for every type stack's $\mathcal{F}_0$. It is used to build the *map types* of morphisms, i.e. the source and target objects correspond to the domains' types and codomains' types, respectively. A combinator with $n > 0$ corresponds to the usual notion of a parameterized (polymorphic) combinator. So if we have morphisms $g_i : A_i \to B_i$ and f is a combinator with the signature of the general form expressed above , we say that $f(g_1, ..., g_n)$ has type $C \to D$ and write $f(g_1, ..., g_n) : C \to D$.

Combinator expressions are inductively defined in the usual manner. We postulate that the identity combinators $id_X : X \to X$ implicitly belong to every combinator base[15]. The set of combinators is also to be closed under instantiations that are consistent with the type stack and under a binary composition operator ".". Note that all compositions will be written in diagrammatic order, a detail that becomes important in the discussion on combinator term reduction.

Examples are again demanded here, so we continue the two earlier examples. First, we have a conventional set of combinators for a CCC[16]

terminal map	$\diamond$	$\mapsto$	$([\,], X \to 1)$
first projection	π_1	$\mapsto$	$([\,], X \times Y \to X)$
second projection	π_2	$\mapsto$	$([\,], X \times Y \to Y)$
product pairing	$\langle _, _ \rangle$	$\mapsto$	$([Z \to X, Z \to Y], Z \to X \times Y)$
curry maps	curry(_)	$\mapsto$	$([X \times Y \to Z], Y \to X \Rightarrow Z)$
evaluation	eval	$\mapsto$	$([\,], X \times X \Rightarrow Z \to Z)$

Second, a set of combinators for manipulating adjunctions is presented:

[15]Note at this point we have to wait for the axiom discussion to determine exactly the implicit behavior of an "identity".

[16]Our notation is somewhat categorical for clarity, even though a programming language environment is intended here by Cockett and Chen.

$$
\begin{array}{rlcl}
\text{left adjoint functor} & L & \mapsto & ([X \to Y], L(X) \to L(Y)) \\
\text{right adjoint functor} & R & \mapsto & ([X \to Y], R(X) \to R(Y)) \\
\text{factorizer} & \lambda & \mapsto & ([X \to R(Y)], L(X) \to Y)) \\
\text{unit} & \beta & \mapsto & ([\,] , X \to R(L(X)))
\end{array}
$$

These examples supports the authors' claim of increased flexibility of their approach over Hagino's system. Now our combinators are not required to be natural transformations and function (type-former) symbols not to be functors.

The last major part of a combinator theory, the axioms, are an explicit set of equational inference rules for the combinators. As we might now anticipate, there is an additional set of rules implicit in every combinator theory. These implicit rules simply state that equality is a substitutive congruence, composition is associative, and the identity combinators behave as identities.

So we now complete the two examples of combinator theories. First, a set of CCC axioms[17]:

$$
\begin{array}{cc}
(\mathrm{term}) & f.\diamond \;=\; \diamond \\
(\mathrm{prod-left}) & \langle f, g\rangle.\pi_1 \;=\; f \\
(\mathrm{prod-right}) & \langle f, g\rangle.\pi_2 \;=\; g \\
(\mathrm{prod-sur}) & \langle x.\pi_1, x.\pi_2\rangle \;=\; x \\
(\mathrm{prod-dist}) & z.\langle x, y\rangle \;=\; \langle z.x, z.y\rangle \\
(\mathrm{exp-inj}) & \langle x, y.\mathrm{curry}(z)\rangle.\mathrm{eval} \;=\; \langle x, y\rangle.z \\
(\mathrm{exp-sur}) & \dfrac{\langle \pi_1, \pi_2.g\rangle.\mathrm{eval} \;=\; z}{\mathrm{curry}(z) \;=\; g}
\end{array}
$$

Now the adjunction axioms:

$$
\begin{array}{ll}
(\mathrm{funct-id}) & L(i) \;=\; i \quad \text{and} \quad R(i) \;=\; i \\
(\mathrm{funct-comp}) & L(x.y) \;=\; L(x).L(y) \quad \text{and} \quad R(x.y) \;=\; R(x).R(y) \\
(\mathrm{fact-inj}) & \beta.R(\lambda(x)) \;=\; x \\
(\mathrm{fact-sur}) & \dfrac{\beta.R(x) \;=\; y}{x \;=\; \lambda(y)}
\end{array}
$$

After choosing a combinator theory appropriate for the computing problem at hand, the categorical programmer must now form a reduction system from the theory. Of course, the axioms must be oriented to form rewriting rules and in many categorical situations, the proper orientation is natural.

However, the reduction system differs in three ways from standard term rewriting reduction: (1) there are notions of *closed* ("well-formed program or arbitrary element") and *canonical* ("minimal program or canonical element") expressions, (2) reduction may be applied to closed terms, and (3) the auxiliary task during reduction of carrying out distributive operations between the main

[17]Not minimal, this set has the redundant (prod-dist) rule. However, we will see later that we need to be aware of the distributivity in such systems.

rewriting steps must be done to supply "outer scopes", or *context*, to all subterms that are intended to have access to it.

The initial step in setting up a reduction system is classifying the combinators into three groups. The *active* combinators are those that cause a rewrite rule to be invoked. More precisely, all rewrite rules will have the form $t.\alpha \Rightarrow r$ where α is an active combinator, t is the prefix (in a canonical form) of α in the overall expression that serves as α's *context*, and the subterm $t.\alpha$ is analogous to a *closure* of context (t) and code (α). The *distributive* combinators are those that satisfy the left distributive rule: $z.\delta(x_1, ..., x_n) = \delta(z.x_1, ..., z.x_n)$. Here z acts as a context that directly affects, by composition, the processing of the arguments of the distributive. The remainder of the combinators are called *constructives*. Usually a constructive can be thought of as a constructor of data that contributes to the final result of a reduction.

In our two examples, the classification results as follows:

CCC		Adjunction
	actives	
π_1, π_2, eval		L, R
	distributives	
$\langle_,_\rangle$, $\diamond$		(none)
	constructives	
curry($_$)		λ, β

We then select corresponding sets of rewrite rules as follows:

CCC	Adjunction
$\langle f, g\rangle.\pi_1 \triangleright f$	$\beta.R(\lambda(x)) \triangleright x$
$\langle f, g\rangle.\pi_2 \triangleright g$	$L(i) \triangleright i$
$\langle x, y.\text{curry}(z)\rangle.\text{eval} \triangleright \langle x, y\rangle.z$	$R(i) \triangleright i$
	$L(x.y) \triangleright L(x).L(y)$
	$R(x.y) \triangleright R(x).R(y)$

At this point we should observe that the reduction rules above, when converted back to equalities, would not necessarily reproduce the original combinator theory. This apparent loss of proof strength should be weighed against the goals of *computing only with the elements* of the theory and of achieving a terminating reduction procedure for the particular problems we wish to solve. These goals should guide this mildly but purposeful ad-hoc selection of actives and orientation of axioms.

We will next outline the general reduction of a closed expression by an abstract machine proposed (and partially implemented [CCS89]) by Cockett, et.al., for computing with categorical combinators. We do so before giving formal definitions and explanations of the reduction theory in order to introduce the intuitive roles of standard forms of combinator expressions used in the processing.

This should aid the reader in understanding the technical definitions that follow. The machine performs the following steps:

- The machine scans the closed expression left-to-right, reducing actives along the main "spine", or composition path, of the expression, creating an expression having only non-actives on its spine. After each active is detected, but prior to matching and applying a rewrite rule to evaluate the closure, the preceding prefix subexpression (termed a *precanonical* because it clearly has no actives with context to operate upon), is traversed right-to-left to collapse any distributives. It is easy to see the collapsing causes the prefix to become a canonical expression by itself.
- The machine then changes direction and scans right-to-left, looking for any distributives on the spine. Possibly new distributives could have been introduced via the rewritings. Nevertheless, if one is found, the entire prefix sub-expression immediately to the left of the distributive, i.e. the distributive's context or outer scope, is distributed to the distributive's arguments. Each of the arguments is then processed as an entire closed expression, each time starting again with the left-to-right mode to reduce actives and finishing with the right-to-left mode to collapse distributives.
- Thus the recursive procedure percolates downward through nestings of distributives with left-to-right passes, transforming the overall expression into a weakly canonical form having only non-actives on its spines and sub-spines (the branching of compositional paths caused by distributives). In this way, all actives that are reachable from the original outermost scope are reduced. Note also that only reachable distributives are being collapsed.
- The machine then eventually percolates upward, making final right-to-left passes to collapse all remaining reachable distributives, and terminates. The resulting expression is canonical.

We now elaborate the forms of combinator expressions.

Definition 5.1 *A closed expression is inductively defined as*

- *Any* 0*-arity distributive is closed.*
- *If the expression has the form* $\delta(e_1, ..., e_n).\gamma_1.\gamma_2.....\gamma_m$ *for* $m \geq 0$ *where* δ *is a distributive, all the* e_i*'s are closed and all the* γ_j*'s are ground (contain no variable parameters), then it is closed.*

Definition 5.2 *A weakly canonical expression is inductively defined as*

- *Any* 0*-arity distributive is weakly canonical.*
- *If the expression has the form* $\delta(e_1, ..., e_n).\gamma_1.\gamma_2.....\gamma_m$ *for* $m \geq 0$ *where* δ *is a distributive, all the* e_i*'s are weakly canonical and all the* γ_j*'s are ground non-actives, then it is weakly canonical.*

Definition 5.3 *A canonical expression is inductively defined as*

- ***Any** 0-**arity distributive is canonical.***
- ***If the expression has the form** $\delta(e_1, ..., e_n).\gamma_1.\gamma_2.....\gamma_m$ **for** $m \geq 0$ **where** δ **is a distributive, all the** e_i**'s are canonical and all the** γ_j**'s are ground constructives, then it is canonical.***

The evaluation strategy is partly top-down and therefore somewhat lazy. The role of the 0-arity distributive should be noted here as well. These distributives, called *terminal combinators* because they are left-absorbing ($f.\delta() = \delta()$), act as λ abstractors and binders to close off the expression from the external context or environment. We see the CCC terminal combinator example of $\diamond$, and we also note the absence of distributors in the adjunction theory. The latter observation simply means we cannot compute with elements involving adjunctions, unless the categories involved contain terminal objects. By the definitions above, canonical terms are weakly canonical, weakly canonical are closed, closed terms are left-absorbing, and from what we have just said, left-absorbing terms are representatives of programs, morphisms, values, or elements.

We now tie together the reduction discussions with a computation example from the CCC combinator theory. The element below reduces to the canonical first projection element of $[X \times Y \Rightarrow X]$.

$$\underbrace{\langle\diamond.\pi_1,\diamond.\pi_2\rangle.\langle\langle\pi_2,\pi_1\rangle,\mathrm{curry}(\pi_1.\pi_2)\rangle}_{\text{precanonical}}.\mathrm{eval}$$

$$\downarrow$$

$$\langle\underbrace{\langle\diamond.\pi_1,\diamond.\pi_2\rangle.\langle\pi_2,\pi_1\rangle}_{\text{weakly canonical}},\langle\diamond.\pi_1,\diamond.\pi_2\rangle.\mathrm{curry}(\pi_1.\pi_2)\rangle.\mathrm{eval}$$

$$\downarrow$$

$$\langle\langle\underbrace{\underbrace{\langle\diamond.\pi_1,\diamond.\pi_2\rangle}_{\text{precanonical}}.\pi_2}_{\text{closure}},\langle\diamond.\pi_1,\diamond.\pi_2\rangle.\pi_1\rangle,\langle\diamond.\pi_1,\diamond.\pi_2\rangle.\mathrm{curry}(\pi_1.\pi_2)\rangle.\mathrm{eval}$$

$$\downarrow$$

$$\langle\langle\diamond.\pi_2,\underbrace{\underbrace{\langle\diamond.\pi_1,\diamond.\pi_2\rangle}_{\text{precanonical}}.\pi_1}_{\text{closure}}\rangle,\langle\diamond.\pi_1,\diamond.\pi_2\rangle.\mathrm{curry}(\pi_1.\pi_2)\rangle.\mathrm{eval}$$

$$\downarrow$$

$$\underbrace{\underbrace{\langle\langle\diamond.\pi_2,\diamond.\pi_1\rangle,\langle\diamond.\pi_1,\diamond.\pi_2\rangle.\mathrm{curry}(\pi_1.\pi_2)\rangle}_{\text{canonical}}.\mathrm{eval}}_{\text{closure}}$$

$$\downarrow$$

$$\underbrace{\underbrace{\langle\langle\diamond.\pi_2,\diamond.\pi_1\rangle,\langle\diamond.\pi_1,\diamond.\pi_2\rangle\rangle}_{\text{canonical}}.\pi_1}_{\text{closure}}.\pi_2$$

$$\downarrow$$

$$\underbrace{\underbrace{\langle\diamond.\pi_2,\diamond.\pi_1\rangle}_{\text{canonical}}.\pi_2}_{\text{closure}}$$

$$\downarrow$$

$$\underbrace{\diamond.\pi_1}_{\text{canonical}}$$

The first active (eval) is reached moving left-to-right. Its precanonical prefix is collapsed moving right-to-left, eventually creating an eval-closure. Note that the first collapse within the prefix creates one argument without any actives to be found moving left-to-right, i.e. it is already weakly canonical. Therefore it can be collapsed immediately into a precanonical form, moving right-to-left. The remaining steps are applications of CCC rewriting rules to closures as the algorithm travels back up the distributive nestings.

The term rewriting underpinnings of the Cockett/Chen methodology are *completeness* and *linearity* of the reduction rules, *deterministic* reduction, and *separability* of canonical expressions. A complete set of reduction rules allows every closure that is legal, or type-correct, to be reducible. To avoid the complexity of matching closures with the left sides of reduction rules in which variables are allowed to occur in both reachable (i.e. with context) and unreachable (i.e. without context) positions, reduction rules must be chosen to be linear (singly-occuring) in each variable. Reducing deterministically means having only one reduction rule applicable to each legal closure. Together determinism and linearity imply uniqueness of a canonical expression if one exists. Finally, separability means that two canonical expressions that are equal outermost syntactically except for unreachable subexpressions (essentially uninvolved in computation) cannot be proved equal in the underlying combinator equational theory. This will permit the characterization of canonicals as the irreducible expressions. With the selection of rewrite rules following these requirements, we say we have a *categorical combinator reduction system.* Thus directly from well-known results in equational systems by Huet and O'Donnell [HL79, O'D77], the authors state the convergence theorem for the abstract combinator machine:

Theorem 5.1 *In a categorical combinator reduction system, reduction is confluent on closed expressions and an expression is irreducible if and only if it is canonical.*

6 What's Next: Control?

This author is particularly interested in pushing category theory even more "externally" towards implementation of an abstract machine that captures control categorically in its varied forms — go-tos, raising exceptions, handling exceptions, while-loops, and more exotic and powerful context-transformation mechanisms such as Felleisen's *prompt* and *control* operators [Fel88, FWFD88, SF90].

A keystone work in this area is Filinksi's SCL-category (SCL for symmetric combinatory logic) that expresses several dualities: totality versus strictness of functions, call-by-name versus call-by-value evaluation, and values versus continuations [Fil89a, Fil89b]. His categorical view of a continuation is a morphism $A \rightarrow 0$ where 0 is a weak initial object. This dualizes the concept of a value as a morphism, or element, $1 \rightarrow A$ where 1 is a weak terminal object. The intuition here is to think of a continuation as an *non-returning* computation[18] that accepts A-typed values.

We also must add that Filinski develops a simply-typed "symmetric" lambda calculus (SLC) that allows equal status for continuations and values. He follows the program of Lambek in developing exact translations in both directions between the SLC and the SCL-category. Thus, in a technically awkward sense, by using these translations Filinski has produced the first abstract machine that

[18] Hence continuations cannot be composed as procedures.

computes with categorical continuations as first-class objects.

The axioms of the SCL-category are listed below. The $\Diamond$ and $\Box$ are the (specified) terminal and initial morphisms, respectively. There are also value variables $x_i : 1 \rightarrow A$, continuation variables $y_i : A \rightarrow 0$, the co-application pa, f^* as the curry of f, and f_* as the co-curry of f. It is important to see that the initial object, terminal object, products, coproducts, exponentials and coexponentials satisfy universality properties weakly, i.e. only the existence of factorizing morphisms is guaranteed. The strange morphisms $\phi : A \times [C \Leftarrow B] \rightarrow [C \Leftarrow A \times B]$ and $\theta : [C \Rightarrow A + B] \rightarrow A + [C \Rightarrow B]$ are primarily for achieving functional completeness that allows the translation from abstractions in the symmetric lambda calculus to the categorical combinators of the SCL. Finally, the SCL condition of totality for the morphism f is $\Diamond \circ f = \Diamond$ where $\Diamond$ is the terminal map[19]. The dual, $f \circ \Box = \Box$, serves as the strictness condition.

primitives are total	primitives are strict
id is total	id is strict
$\circ$ of totals is total	$\circ$ of stricts is strict
$\Diamond$ is total	$\Diamond$ is strict
$\langle _, _ \rangle$ of totals is total	$\langle -, - \rangle$ of stricts is strict
π_i is total	π_i is strict
$\Box$ is total	$\Box$ is strict
$[_, _]$ of totals is total	$[_, _]$ of stricts is strict
ι_i is total	ι_i is strict
ϕ is total	ϕ is strict
θ is total	θ is strict
x_i is total	y_i is strict
pa is total	ap is strict
f^* is total	f_* is strict

$$\Diamond_1 = id_1 \qquad\qquad \Box_0 = id_0$$

$$\frac{g \text{ is total}}{\pi_1 \circ \langle f, g \rangle = f} \qquad\qquad \frac{g \text{ is strict}}{[f, g] \circ \iota_1 = f}$$

$$\frac{f \text{ is total}}{\pi_2 \circ \langle f, g \rangle = g} \qquad\qquad \frac{f \text{ is strict}}{[f, g] \circ \iota_2 = g}$$

$$\langle \pi_1, \pi_2 \rangle = id \qquad\qquad [\iota_1, \iota_2] = id$$

[19] Interestingly, this same representation of totality had been earlier used in the partial cartesian categories of Curien and Obtulowicz.

$$\frac{h \text{ is total}}{\langle f,g\rangle \circ h = \langle f \circ h, g \circ h\rangle} \qquad \frac{h \text{ is strict}}{h \circ [f,g] = [h \circ f, h \circ g]}$$

$$\frac{h,k \text{ are total}}{(h \times k) \circ \langle f,g\rangle = \langle h \circ f, k \circ g\rangle} \qquad \frac{h,k \text{ are strict}}{[f,g] \circ (h+k) = [f \circ h, g \circ k]}$$

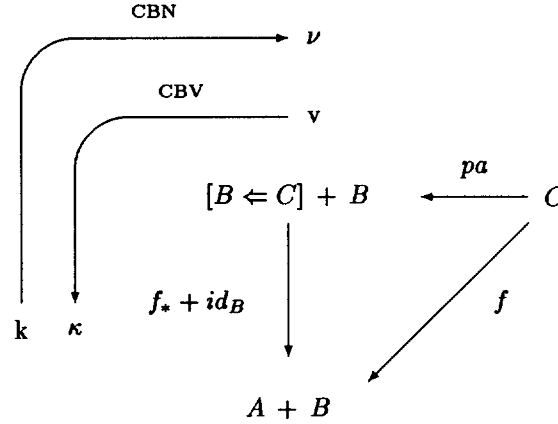
$$ap \circ (f^* \times id) = f \qquad (f_* + id) \circ pa = f$$

$$\frac{f \text{ is total}}{(ap \circ (f \times id))^* = f} \qquad \frac{f \text{ is strict}}{((f + id) \circ pa)_* = f}$$

$$\phi \circ \langle f \circ \Diamond, id\rangle = (pa \circ \langle f \circ \Diamond, id\rangle)_* \quad [\Box \circ f, id] \circ \theta = ([\Box \circ f, id] \circ ap)^*$$

$$(f \circ \pi_2)_* \circ \theta = f_* \circ \pi 2 \qquad \theta \circ (\iota_2 \circ f)^* = \iota_2 \circ f^*$$

There are some suggestions about control within the SCL category if Filinski's CBV denotational *morphism* semantics are followed. For example, consider how the coapplication *pa invokes* the cocurry f_* in a manner dual to the invoking of the curry f^* with the application *ap*. The allowed strategies of evaluation are illustrated below. Note that CBV evaluates the morphisms in diagrammatic order while CBN evaluates them in compositional order. Here v and k represent the evaluated value and continuation, and ν and κ the unevaluated value and continuation.



The applicable semantics are

$$\begin{aligned} \mathcal{M}[\![f \circ g]\!]v\kappa &= \mathcal{M}[\![g]\!]v(\lambda t.\, \mathcal{M}[\![f]\!]t\kappa) \\ \mathcal{M}[\![pa]\!]v\kappa &= \kappa\, in_1(contx(v, \lambda t.\, \kappa\, in_2(t))) \\ \mathcal{M}[\![f_*]\!]v\kappa &= \text{let } contx(a,c) = v \text{ in } \mathcal{M}[\![f]\!]a(\lambda t.\text{case } t \text{ of } in_1(r) : \kappa\, r \,[\!]\, in_2(s) : c\, s \text{ esac}) \end{aligned}$$

Following these SCL CBV semantics, the elements of the coexponential $[B \Leftarrow C]$ are continuations that accept a C-value/B-accepting continuation pair. These elements are termed as *context* values. The computation proceeds in detail as follows:

1. *pa* injects (tags) the context pair (c, k_B) into $[B \Leftarrow C] + B$, where c is a C-value and k_B is a B-accepting *bridging* continuation.

2. Control then passes to the $[B \Leftarrow C] + B$-accepting continuation $k_{[B \Leftarrow C]+B}$ following *pa*, i.e.

the continuation starting with $f_* + id_B$. As a result, $f_* + id$ runs and sees its input tagged as a context, viz. (c, k_B), allowing only f_* to process it.

3. If $f_*(c)$ is in A, then the continuation k_{A+B} following $f_* + id$ computes the answer from $f_*(c)$.
4. Otherwise $f_*(c)$ is in B, so f_* *returns* to pa with the value $f_*(c)$.
5. pa then executes the bridging continuation k_B. This continuation first injects $f_*(c)$ into the B summand of $[B \Leftarrow C] + B$.
6. Then k_B re-executes the $k_{[B \Leftarrow C]+B}$ continuation. Again $f_* + id$ runs and sees only the injected $f_*(c)$, causing id_B to pass $g_*(c)$ directly on to the k_{A+B} continuation to compute the answer.

What this all amounts to is:

- The pa processing sets up a *bridging* continuation for the B (f_* result) case, i.e. it "suspends" the B-related continuation prior to the cocurry processing.
- The cocurry processing sets up the continuation for the A (f_* result) case.
- The cocurry processing does all the tag checking.
- The cocurry processing *backtracks* from the $[B \Leftarrow C]$-accepting sub-continuation of $k_{[B \Leftarrow C]+B}$ via the bridging continuation to the B-accepting sub-continuation if a B tag of the f_*-result is detected.
- Rephrasing the preceding point, pa requests that f_* "try" the "A-related" continuation first, and if f_* cannot continue, "does" the "B-related" continuation.

The difference between currying and cocurrying can now be expressed in a new way. In currying, all the parameter values — evaluated one at a time — must be applied to by the function, while in cocurrying the parameter continuations must be successively *attempted* with only one being selected. Thus the cocurry side of the composition of pa and $f_* + id_B$ provides the "initial" continuation, and the co-application side provides the *ordering of the attempts* and the "bridge" to the alternative continuation.

Also, the co-application morphism does no tag checking, or performs any other discriminatory functions.

Another intuition that is consistent with the Filinski semantics is to consider the pa morphism as a CBV-version of "call with exception propagate". The call and exception return state is represented by the co-exponential $[B \Leftarrow C]$ and the exception propagation mechanism is represented by the bridging to the B summand and id_B. Thus f_*'s tag checking corresponds to exception-detection and a backtrack to pa "raises" the exception.

But the Filinski result is only an initial step towards categorical computation using *control morphisms*. There are serious shortcomings in attempting to use an SCL-category for reasonable *direct* computation.

First, the SCL-category is quite minimal with respect to the three dualities stated earlier. In particular, the SCL definition of a continuation morphism is one-ended or *global*, while most control

constructs are two-ended or *local*: a new context — a value-continuation pair — is computed in some manner ***in terms of the current context*** to replace the current context. Also, the concept of a continuation that "goes on forever, never returning", i.e. without knowledge of if and when it will end, seems overly simplified for the sake of categorical and semantic convenience. The general need to raise, propagate, and handle exceptions demands a *local* or *bounded* continuation that is two-ended and *composable* in nature. Support for such "procedural" or "operator" continuations has come forth not only from Felleisen, who provides a dynamic semantics for his control operators, but also in new work by Danvy and Filinski that establishes a static semantics for them as well [DF90], and by Griffin who showed that a simpler but very expressive form of the Felleisen context-shifting operator — which possesses a *classical* type — has a precise interpretation in constructive logic [Gri90].

Second, the axioms of the category are conditionally based on the totality or strictness of morphisms. This negates the possibility of a Cockett-style term rewriting reduction. One might consider applying conditional term rewriting theory, including Knuth-Bendix completion, to an SCL-category [Kap84, Kap87, Gan87]. However, (1) as claimed above, the SCL-category appears not rich enough to be computationally useful in the first place (an opinion shared by Filinski [Fil90]), and (2) severe constraints on the premises of conditional rules are required to avoid undecidability. It is for this reason that categories possessing weak products/coproducts — such as the SCL-category — along with a more sophisticated and fully equational structure similar to the p-category of Robinson and Rosolini might hold promise as a new venue for computing with control morphisms.

Third, computation is defined not entirely by the equational logic of the SCL category — the axioms are strategy-independent — but by two sets of denotational semantics, one for CBV and the other for CBN. This represents both a severe coarseness in the strategy spectrum and a weakness in the direct programmability of the SCL category. Although there are continuations in the category, they play little direct role in the control of the categorical computation itself.

Each of the papers principally surveyed in this report has influenced our pondering on the possibilities for embedding a useful set of control constructs within a category. Here are some examples. Could local continuations be internalized as some kind of "control-closed" category or a "control-enriched" category? Is a p-category, which shares many similar properties as the SCL category, a starting point for including control? Does the dual of the p-category's *dom*-operator have any relationship to control? If such a category is found, could the Cockett methodology be extended, if necessary, to deal with actives that have contexts either as prefixes (evaluation of values) or postfixes (evaluation of continuations), as a technique to include control operations?

And what about other categories? One example of possible relevance is the *Girard category* that is essentially a closed symmetric monoidal category with tensor products and exponentials, finite categorical products and sums, and an *involution* functor that intuitively corresponds to converting back and forth between values and continuations [See89]. In fact, a Girard category is a special brand of *-automonous category [Bar79]. This category contains both weak and categorical products and sums depending on the "non-strictness" and "non-totality" of arguments, properties similar to those of the SCL category. It has been proved by Seely to correspond directly to Girard's linear logic [Laf88, Sce90], which in turn has been shown to fit hand-in-glove with computation using Petri nets [MOM89, EW90]. Since this relationship has strong implications for parallel computation, the Girard category is so far only mildly promising for categorizing control.

Another farther-fetched possibility is higher-order categories such n-categories. Seely has expressed the typed lambda calculus as a 2-category where the 2-cells express beta/eta-reductions of terms with a single free variable [See87]. Carrying the analogy farther, a higher-order cell might express some kind of reduction transformation that switches contexts and exchanges continuations, with each cell level representing the next-higher local context of the current computation.

7 Acknowledgement

The author wishes to thank Edmund Robinson and James Hook for their critical review of an early draft and excellent recommendations for improvements in this presentation.

References

[AL86] A. Asperti and G. Longo, *Categories of Partial Morphisms and the Relation between Type-Structures*, Nota Scientifica S-7-85, Dipartimento di Informatica, Università di Pisa, 1985.

[Bar79] **-Autonomous Categories*, Lecture Notes in Mathematics 752, 1979.

[Bar90] M. Barr, Fixed Points in Cartesian Closed Categories, *Theoretical Computer Science 70*, 1990.

[BW85] M. Barr and C. Wells, *Toposes, Triples and Theories*, Springer-Verlag, 1985.

[BW90] M. Barr and C. Wells, *Category Theory for Computing Science*, Prentice-Hall, 1990.

[CC90] J. R. B. Cockett and H. G. Chen, *Categorical Combinators*, Preprint, May, 1990.

[CCS89] J. R. B. Cockett, H. G. Chen, L. R. Chen and L. R. Smith, *Preliminary Users Manual for* **Charity**, Technical Report CS-89-82, University of Tennessee, 1989.

[CO89] P.-L. Curien and A. Obtulowicz, Partiality, Cartesian Closedness, and Toposes, Information and Computation 80, No. 1, 1989.

[Cur86] P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming*, John Wiley, 1986.

[DF90] O. Danvy and A. Filinski, Abstracting Control, In *Proc. ACM Lisp and Functional Programming Conference*, 1990.

[DH87] R. A. Di Paola and A. Heller, Dominical Categories: Recursion Theory without Elements, *J. Symbolic Logic 52*, 3, 1987.

[EW90] U. Engberg and G. Winskel, Petri Nets as Models of Linear Logic, In *Proc. CAAP*, Lecture Notes in Computer Science 431, 1990.

[Fre74] P. J. Freyd, Allegories, mimeographed notes.

[FS90] P. Freyd and A. Scedrov, *Categories, Allegories.* Forthcoming book, North-Holland, 1990.

[Fel88] M. Felleisen, The Theory and Practice of First-Class Prompts. In *15th Symposium on Principles of Programming Languages*, 1988.

[Fil89a] A. Filinski, Declarative Continuations: An Investigation of Duality in Programming Language Semantics, *Summer Conference on Category Theory and Computer Science*, Lecture Notes in Computer Science 389, 1989.

[Fil89b] A. Filinski, *Declarative Continuations and Categorical Duality*, Master's thesis, University of Copenhagen, 1989.

[Fil90] A. Filinski, private communication, 1990.

[FWFD88] M. Felleisen, M. Wand, D. Friedman, and B. Duba, Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps. In *ACM Conference on Lisp and Functional Programming*, 1988.

[Gan87] H. Ganzinger, A Completion Procedure for Conditional Equations. In *1st Intl. Workshop Conditional Term Rewriting Systems*, Lecture Notes in Computer Science 308, 1987.

[Gri90] T. Griffin, A Formulae-as-Types Notion of Control. In *Symposium on Logic in Computer Science*, 1990.

[GMS89] C. Gunter, P. Mosses, and D. Scott, *Semantic Domains and Denotational Semantics*, Technical Report MS-CIS-89-16, University of Pennsylvania, 1989.

[Hag87] T. Hagino, *A Categorical Programming Language*, Ph. D. thesis, University of Edinburgh, 1987.

[HL79] G. Huet and J-J. Levy, Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems, INRIA, Technical Report 359, 1979.

[HP86] H. Huwig and A. Poigne, A Note on the Inconsistencies Caused by Fixpoints in a Cartesian Closed Category, *Theoretical Computer Science 73*, 1990.

[Jay90] C. Barry Jay, *Extending Properties to Categories of Partial Maps*, Technical Report ECS-LFCS-90-107, University of Edinburgh, 1990.

[Kap84] S. Kaplan, Conditional Rewrite Rules, *Theoretical Computer Science 33*, 1984.

[Kap87] S. Kaplan, Simplifying Conditional Term Rewriting Systems: Unification, Termination and Confluence, *J. Symbolic Computation 4*, 1987.

[Laf88] Y. Lafont, *Introduction to Linear Logic*, Summer School on Constructive Logics and Category Theory, 1988.

[Lam86] J. Lambek, Cartesian Closed Categories and Typed λ-Calculi, In *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science 242, Springer-Verlag, 1986.

[LM84] G. Longo and E. Moggi, Cartesian Closed Categories of Enumerations for Effective Type Structures. In *Intl. Symposium of Data Types*, Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[Mac71] S. MacLane, *Categories for the Working Mathematician*, Springer-Verlag, 1971.

[Mog86] E. Moggi, Categories of Partial Morphism and the λ_p-calculus, In *Category Theory and Computer Programming*, Lecture Notes in Computer Science 240, Springer-Verlag, 1986.

[Mog88a] E. Moggi, *Computational Lambda-Calculus and Monads*, Technical Report ECS-LFCS-88-66, University of Edinburgh, October, 1988.

[Mog88b] E. Moggi, Partial Morphisms in Categories of Effective Objects, *Information and Computation 76*, 1988.

[Mog89a] E. Moggi, Computational Lambda-Calculus and Monads. In *Proceedings of the 16th Symposium on Principles of Programming Languages*, Austin,Texas, 1989.

[Mog89b] E. Moggi, Lecture notes on *An Abstract View of Programming Languages*, July, 1989.

[Mog90] E. Moggi, *Notions of Computations and Monads*, Preprint, April, 1990.

[MOM89] N. Martí-Oliet and J. Meseguer, From Petri Nets to Linear Logic, In *Category Theory in Computer Science*, Lecture Notes in Computer Science 389, 1989.

[O'D77] M. O'Donnell, *Computing in Systems Described by Equations*, Lecture Notes in Computer Science 58, 1977.

[Pie90] B. Pierce, *A Taste of Category Theory for Computer Scientists*, Technical Report CMU-CS-90-113, Carnegie Mellon University, 1990.

[Rom89] L. Román, Cartesian Categories with Natural Numbers Object, *J. Pure Appl. Alegbra,* 1989.

[Ros86] G. Rosolini, *Continuity and Effectiveness in Topoi*, D. Phil. thesis, University of Oxford, 1986.

[RR88] E. Robinson and G. Rosolini, Categories of Partial Maps, *Information and Computation, 79*, 1988.

[Sce90] A. Scedrov, A Brief Guide to Linear Logic, *Bulletin of the EATCS 41*, June, 1990.

[See87] R. Seely, Modelling Computations: A 2-Categorical Approach. In *Symposium on Logic in Computer Science*, 1987.

[See89] R. Seely, Linear Logic, *-autonomous Categories and Cofree Coalgebras, *Contemporary Mathematics 92*, 1989.

[SF90] D. Sitaram and M. Felleisen, Control Delimiters and their Hierarchies, *Lisp and Symbolic Computation 3*, 1990.

[Sri90] Y. V. Srinivas, *Category Theory: Definitions and Examples,* Technical Report 90-14, University of California, Irvine, February, 1990.

[Wra89] G. C. Wraith, A Note on Categorical Data Types. In *Category Theory in Computer Science*, Lecture Notes in Computer Science 389, 1989.