Linear Continuations *

Andrzej Filinski

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
andrzej+@cs.cmu.edu

Abstract

We present a functional interpretation of classical linear logic based on the concept of *linear continuations*. Unlike their non-linear counterparts, such continuations lead to a model of control that does not inherently impose any particular evaluation strategy. Instead, such additional structure is expressed by admitting closely controlled copying and discarding of continuations. We also emphasize the importance of classicality in obtaining computationally appealing categorical models of linear logic and propose a simple "coreflective subcategory" interpretation of the modality "!".

1 Introduction

In recent years, there has been considerable interest in applications of Girard's Linear Logic (LL) [Gir87] to programming language design and implementation. Over time, various more or less mutated versions of the original system have been proposed, but they all share the same basic premise: that assumptions made in the course of a formal proof can not necessarily be used an arbitrary number of times – including zero – in deriving the conclusion. This idea in itself is not new; numerous formalizations of "relevant implication" are known, and even Church's original λI-calculus required the abstracted variable to occur free at least once in the body of an abstraction. In (canonical) LL this restriction is strengthened even further: assumptions must be used *exactly once*. However, a modal construct is also added for representing hypotheses that may be used without restrictions. A striking consequence of this "resource-consciousness" is that the otherwise problematic *negation* can be made an involution, *i.e.*, "not not A" and A can be identified, without loss of constructivity.

The advantages of making linearity explicit in functional languages have been convincingly argued before [Laf88a, Hol88, Wad90]. The primary benefits cited are that a linear framework can naturally express single-threadedness (hence transparent destructive updates), and reduce or even eliminate the need for garbage collection. This stands in contrast to deducing such information after the fact by program analysis techniques such as abstract interpretation. Instead, the program can *explicitly* state that a piece of data will always be accessible through a single reference, and that attempts to copy or discard it should be considered type errors. Apart from making the intent clearer, such an approach enhances modularity and simplifies some issues of separate compilation: linearity can be made part of the visible interface instead of being deduced from the (possibly still non-existent) implementation.

Notably, however, such applications exploit only the *intuitionistic* or negation-less fragment of LL. They essentially formalize a notion of *linear data*, which is used exactly once unless otherwise specified. Yet there remains a potentially even more significant application of keeping closer track of resources: *linear control*, where the process of evaluation itself is made explicit and subject to linearity constraints. As we will see, *classical* linear logic permits the control flow of a language (including the evaluation strategy) to be modeled naturally using the same basic concepts as those used for expressing its data flow. In fact, the fundamental idea is precisely to reflect the domain of control into the world of linear data using the tool of *first-class linear continuations*.

Again, this approach complements implementation techniques such as strictness analysis: instead of treating sites of "exactly-once" evaluation as isolated islands

*This research was sponsored in part by National Science Foundation Grant CCR-8922109 and in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

(To appear in POPL'92)

of linearity in a fundamentally non-linear language, a theory of linear control views linear constructs as the very skeleton around which non-linear features are built. Perhaps somewhat paradoxically, a purely linear evaluation framework does not limit parallelism or overspecify an evaluation strategy – even in the presence of first-class continuations! Only when non-linear constructs must be accommodated (for example, when pieces of residual computation must be discardable without any knowledge of their internal structure) does the model impose a definite evaluation order.

Since the proper formalization of linearity – especially for control – is not always intuitively obvious, we will rely on category theory as an organizational tool and guiding light. In particular, we will be formalizing linear control in the setting of *linear categories*, a close relative of the ubiquitous cartesian closed categories (CCCs) permeating most work on categorical semantics of programming languages; essentially, linear categories are to (classical) LL what CCCs are to intuitionistic logic. At all times, however, we will try to support the categorical definitions and results by computational intuition.

The paper is organized as follows: section 2 presents linear continuations in a categorical setting and outlines their role in tying together formally dual constructs. Section 3 considers categorical interpretations of the modal operators of LL and the significance of classicality in obtaining simple models. Section 4 outlines how classical LL can be used to faithfully represent the data types and evaluation strategies of functional languages by varying the interpretation of negation. Finally, section 5 discusses connections with related work, and section 6 summarizes the main ideas and suggests some promising areas for further investigation.

2 Linear Categories

This section reviews the by now common interpretation of modality-free classical linear logic as a linear category and introduces the concept of first-class linear continuations as a paradigm for reasoning about functional interpretations of such categories. The reader unfamiliar with category theory should probably consult the appendix for an introduction to the terminology used.

2.1 Tensor products and linear exponentials

The usual categorical presentations of typed λ-calculi build on the concept of *cartesian closure* [LS86], often supplemented with rich additional structure to express refinements like polymorphism or dependent types. But at the very core of all such systems remains the intuitionistic principle that all available data at a given point can be used any number of times, and possibly not at all. This is a reasonable assumption in many cases, but it excludes a number of otherwise very useful categorical models which possess only the weaker property of *symmetric monoidal closure*, corresponding to *linear λ-calculi*. As a first step towards defining such categories and calculi, we formalize the fundamental concept of aggregating single-use data:

Definition 1 *A* symmetric monoidal category *is a category $\mathcal{C}$ equipped with a distinguished object 1 ("unit"), a bifunctor $- \otimes - : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ ("tensor product"), and natural isomorphisms*

$$\begin{aligned} &\mathbf{delr}_A : A \otimes 1 \to A \\ &\mathbf{assl}_{A,B,C} : A \otimes (B \otimes C) \to (A \otimes B) \otimes C \\ &\mathbf{exch}_{A,B} : A \otimes B \to B \otimes A \end{aligned}$$

satisfying the Mac Lane-Kelly coherence conditions *[Mac71].*

(The coherence conditions formalize the expected relations among the "structural isomorphisms" above. For example, the identity $\mathbf{exch}_{B,A} \circ \mathbf{exch}_{A,B} = \mathbf{id}_{A \otimes B}$ does not follow automatically from naturality of **exch**, but is guaranteed by coherence.)

In particular, any cartesian category is symmetric monoidal, with the terminal object as unit and the categorical product as tensor product. But often the "natural" product in a category is only a tensor product, not a categorical one. For example, consider the category $\mathbf{Set}_p$ of sets and partial functions with a designated one element-set as unit and the *cartesian* product of two sets as their tensor product; the tensor product of two partial functions f and g is the function $f \otimes g = (a,b) \mapsto (f(a), g(b))$, defined iff $f(a)$ and $g(b)$ are both defined. Such products are different from the *categorical* products in $\mathbf{Set}_p$, where, *e.g.*, the empty set is terminal.

Even though tensor products lack the projections and pairings characterizing their categorical counterparts, we can still define exponentials in the usual way by exploiting the functoriality of $\otimes$:

Definition 2 *A symmetric monoidal category is called* closed *if the functor $- \otimes B$ has a right adjoint $B \multimap -$ ("linear exponential") for every object B. We will often use the abbreviation* SMCC *for such a category.*

We write the counit of the adjunction as **ap** ("application") and the factorizer as **cur**$(-)$ ("currying"), giving the two adjunction equations:

$$\begin{aligned} \mathbf{ap}_{B,C} \circ (\mathbf{cur}(f) \otimes \mathbf{id}_B) &= f : A \otimes B \to C \\ \mathbf{cur}(\mathbf{ap}_{B,C} \circ (g \otimes \mathbf{id}_B)) &= g : A \to B \multimap C \end{aligned}$$

(Following the conventions for ordinary products and exponentials, the operator $\otimes$ binds tighter than $\multimap$.)

For example, $\mathbf{Set}_p$ with the tensor product mentioned above is symmetric monoidal closed. It is not *cartesian closed*, however – the categorical product in $\mathbf{Set}_p$ does not have a right adjoint.

Analogously to cartesian closed categories (CCCs) and λ-calculi, we can express SMCC morphisms using a more concise lambda-syntax extended with a tuple notation for tensor products. The crucial property of such a *linear λ-calculus* is that data must be used exactly once, leading naturally to variable patterns in λ-abstractions [Laf88a]. For example, the linear λ-term with a free variable $a : A$

$$\lambda(b^B, f^{A \otimes B \multimap C}).\, f\,(a, b)$$

represents a categorical morphism with source A:

$$\mathbf{cur}(\mathbf{ap}_{A \otimes B, C} \circ \mathbf{exch}_{A \otimes B, A \otimes B \multimap C} \circ \mathbf{assl}_{A,B,A \otimes B \multimap C}) : A \to B \otimes (A \otimes B \multimap C) \multimap C$$

The adjunction equations correspond to the usual $\beta\eta$-equivalence between terms, just as for ordinary λ-calculi and cartesian closure.

We can also note that the product-free fragment of the linear λ-calculus corresponds to a combinatory logic in which the usual basic combinators $S = \lambda fgx.\, fx(gx)$, $K = \lambda xy.\, x$ (and $I = \lambda x.\, x = SKK$) are replaced by the *linear* combinators $B = \lambda fgx.\, f(gx)$, $C = \lambda fxy.\, fyx$, and $I = \lambda x.\, x$. These do indeed use each abstracted variable exactly once, unlike both S (which uses x twice) and K (which discards y).

2.2 Duality and negation

As outlined above, the categorical interpretation of *intuitionistic* LL as an SMCC provides a semantical basis for higher-order functions over linear data. We will now see how the negation operator of *classical* LL can be interpreted as extending the internal language of the category with a construct for reasoning about *control as data*. To this end, we will consider the very natural categorical characterization of classicality in terms of dualizing objects, due to Martí-Oliet and Meseguer [MOM90, MOM91]. Despite its apparent simplicity, it is actually slightly stronger than the interpretation based on *-autonomous categories [Bar79] outlined by Seely [See89].

For any object C in an SMCC, there exists a natural (in A) transformation $A \to (A \multimap C) \multimap C$; we can think of it as the function $\lambda a^A.\, \lambda k^{A \multimap C}.\, k\, a$. When this function is invertible for every A, we have:

Definition 3 *An object $\bot$ in an SMCC is called a* dualizing object *if the natural transformation* $\mathbf{cur}(\mathbf{ap}_{A,\bot} \circ \mathbf{exch}_{A, A \multimap \bot}) : A \to (A \multimap \bot) \multimap \bot$ *is a (natural) isomorphism. An SMCC with a dualizing object will be called* classical.

We will write the required inverse morphism as $\mathbf{eval}_A : (A \multimap \bot) \multimap \bot \to A$, giving the pair of equations

$$\begin{aligned} \mathbf{eval}_A \circ \mathbf{cur}(\mathbf{ap}_{A,\bot} \circ \mathbf{exch}_{A, A \multimap \bot}) &= \mathbf{id}_A \\ \mathbf{cur}(\mathbf{ap}_{A,\bot} \circ \mathbf{exch}_{A, A \multimap \bot}) \circ \mathbf{eval}_A &= \mathbf{id}_{(A \multimap \bot) \multimap \bot} \end{aligned}$$

We note in particular that $\bot \multimap \bot \;\cong\; (1 \multimap \bot) \multimap \bot \cong 1$. The combination $A \multimap \bot$ is usually abbreviated $A^\bot$, the *linear negation* of A. (Syntactically, $\cdot^\bot$ binds tighter than any prefix or infix operator; the notation symbolizes "A orthogonal", not a function space.)

It would have been very convenient to have such a negation in a cartesian closed category. Unfortunately, it is fairly easy to show (*e.g.*, [LS86, p. 67]; the observation is usually credited to Joyal) that a CCC cannot have a dualizing object unless it is a preorder, *i.e.*, contains at most one morphism between any two objects. We thus really need the additional restrictions imposed by linearity to obtain non-degenerate categories with dualizing objects. For example, the category **Cohl** of coherent spaces and linear maps [Gir87] (which has independent semantic interest) is actually a classical SMCC [See89, MOM90].

We can introduce the counterpart of **eval** (no relation to the Lisp construct of the same name) in the linear λ-calculus as a special operator $\mathcal{D}$ with typing rule

$$\frac{\Gamma \vdash E : (A \multimap \bot) \multimap \bot}{\Gamma \vdash \mathcal{D}\, E : A}$$

The categorical isomorphism equations then correspond directly to the following two identities between linear λ-terms:

$$\begin{aligned} \mathcal{D}\,(\lambda k^{A \multimap \bot}.\, k\, E) &= E : A \\ \lambda k^{A \multimap \bot}.\, k\,(\mathcal{D}\, E) &= E : (A \multimap \bot) \multimap \bot \end{aligned}$$

What is the computational intuition behind these equalities? If we think of a value of type $1 \multimap A$ as representing a function with no input, a value of the dual type $A \multimap \bot$ would logically represent a function with no output; we will call such a "black hole" function a *(linear) A-accepting continuation.*

With this view, $\mathcal{D}$ becomes a construct for adding *first-class* (*i.e.*, usable with the same generality as other values) continuations to the language: $\mathcal{D}$ calls its argument E with the current continuation represented as a non-returning function. Because of linearity constraints, E must eventually apply this continuation to some A-typed value[1], and computation resumes "as if" $\mathcal{D}$ had just returned that value. In other words, $\mathcal{D}$ allows us to *reify* an otherwise intangible evaluation context into a piece of data.

[1] We assume here that the "initial continuation" is itself linear, *i.e.*, that the final result returned by the program will actually be "used up". In particular, if the result contains functional components, each of those will also be applied exactly once to a value of the appropriate type.

The first equation above can now be paraphrased as "capturing the current continuation k and immediately applying it to E (*i.e.*, abstracting the current context, then evaluating E in that context) is redundant." The second, which can be equivalently stated as $k\,(\mathcal{D}\,E) = E\,k$, tells us that "when a continuation k is applied to $\mathcal{D}\,E$ (*i.e.*, when $\mathcal{D}\,E$ is evaluated in a context represented by k), that continuation will be captured by $\mathcal{D}$ and passed to E as an argument."

Seen in this way, $\mathcal{D}$ closely resembles Scheme's `call/cc` [RC86], the $\mathcal{C}$-operator of [FFKD87], or the version of $\mathcal{C}$ considered in [Gri90] (where the idea of double-negation elimination as a control operator was first presented, but in a non-linear setting). In particular, $\mathcal{D}$ satisfies equations

$$\begin{aligned} E'\,(\mathcal{D}\,E) &= \mathcal{D}\,(\lambda k.\,k\,(E'\,(\mathcal{D}\,E))) \\ &= \mathcal{D}\,(\lambda k.\,[\lambda a.\,k\,(E'\,a)]\,(\mathcal{D}E)) \\ &= \mathcal{D}\,(\lambda k.\,E\,(\lambda a.\,k\,(E'\,a))) \\ (\mathcal{D}\,E)\,E' &= \mathcal{D}\,(\lambda k.\,k\,((\mathcal{D}\,E)\,E')) \\ &= \mathcal{D}\,(\lambda k.\,[\lambda f.\,k\,(f\,E')]\,(\mathcal{D}E)) \\ &= \mathcal{D}\,(\lambda k.\,E\,(\lambda f.\,k\,(f\,E'))) \end{aligned}$$

analogous to Felleisen's rules for the $\mathcal{C}$-operator, which allow us to "bubble" $\mathcal{D}$s up towards the root of a term.

But the properties of a dualizing object have wider-ranging consequences than might be expected. Essentially, linear continuations *commute*, ensuring a kind of coherence among linear λ-terms that is not guaranteed by the SMCC axioms alone. For example, given two "linear request-acceptors" $E_1 : A^{\perp\perp}$ and $E_2 : B^{\perp\perp}$, we can obtain a value of type $A \otimes B$ in two different ways depending on which expression we "query" first, but we easily check that

$$\begin{aligned} &\lambda k.\,E_1\,(\lambda a.\,E_2\,(\lambda b.\,k\,(a,b))) \\ &\quad = \lambda k.\,E_2\,(\lambda b.\,E_1\,(\lambda a.\,k\,(a,b))) \\ &\quad = \lambda k.\,k\,(\mathcal{D}\,E_1, \mathcal{D}\,E_2) \end{aligned}$$

In other words, "linear continuation-passing style" does not inherently pick an arbitrary evaluation order among tuple or application components; such an order is only imposed when the continuations can be discarded or invoked multiple times.

This property may even provide a link between "true" parallelism and continuation-based multiprogramming [Wan80]. In the latter, a `call/cc`-like operator is essentially used to implement a *coroutine* facility, where captured continuations are never applied twice. Each "thread of control" makes independent progress without backtracking, and first-class continuations are used only to multiplex them onto a single processor. This linear use of continuations is especially significant for preemptive (timer-based) scheduling, where continuations can be captured and switched asynchronously during execution of what would otherwise appear as ordinary, sequential code.

Moreover, if we take only negation (suitably axiomatized) and tensor products as primitives, we can *define* the linear exponential $A \multimap B$ as $(A \otimes B^{\perp})^{\perp}$. In the computational interpretation, we can thus think of a functional value as a continuation accepting a pair ("application context") consisting of an A-typed argument and a B-typed return continuation. Linearity ensures that the return continuation is eventually invoked, so all linear functions do return to their place of call. This view of functions as continuations will be particularly useful later, when we replace linear negation with *call-by-value negation* to get a traditional Scheme-like language where functions need not return exactly once.

2.3 Additives

Linear logic also defines a set of "additive connectives" which correspond to categorical products and coproducts. In the intuitionistic subset they must be axiomatized separately, but it is easily shown that if a classical SMCC has a terminal object $\top$ and binary products $A \,\&\, B$, it also has an initial object 0 and binary coproducts $A \oplus B$, and vice versa; such a category is called *linear*. Computationally, a value of type $A \,\&\, B$ represents a pair of which *either* the A-typed *or* the B-typed component must be used in the computation, but the choice can be made at the point of use. Conversely, a value of type $A \oplus B$ is a value either of type A or of type B, the choice being made at the point of creation.

For reasons which will become apparent in the next section, we choose to treat the coproducts as fundamental and *define* $A \& B$ as $(A^{\perp} \oplus B^{\perp})^{\perp}$. It is instructive to note how this equivalence is explained very naturally in terms of continuations: a product of A and B is interpreted as a continuation accepting *tagged requests* for either an A or a B. This is reflected by the projection and pairing morphisms:

$$\begin{aligned} \mathbf{fst}(p) &= \mathcal{D}\,(\lambda k^{A^{\perp}}.\,p\,(\mathbf{inl}(k))) \\ \mathbf{snd}(p) &= \mathcal{D}\,(\lambda k^{B^{\perp}}.\,p\,(\mathbf{inr}(k))) \\ \langle f, g\rangle(c) &= \lambda r^{A^{\perp} \oplus B^{\perp}}.(\mathbf{case}\ r\ \mathbf{of}\ \mathbf{inl}(k_1) \rightarrow k_1\,(f(c))\ [\!]\ \mathbf{inr}(k_2) \rightarrow k_2\,(g(c))) \end{aligned}$$

The product equations follow directly from the ones for the coproduct and negation. Similarly, we let $\top = 0^{\perp}$, an object from which no information can be extracted.

3 Modal Operators

The fragment of linear logic presented so far is not very expressive. For general computation, we need a way to express non-linear uses of data, such as Girard's modal operators "!" and "?". Unfortunately, their semantic

properties are not nearly as well understood as the pure linear fragment; in particular, the proposed categorical interpretation of "!" [Laf88a] is often considered somewhat controversial (*e.g.*, [See89]).

One reason for these problems may be that in the intuitionistic fragment of LL, "!" is forced to perform two rather unrelated functions: expressing copyability/discardability of data and suspending potentially non-terminating computations. In this section, we will see how the additional expressive power provided by full (*i.e.*, classical) linear logic and the modality "?" allows us to properly separate these concepts. More precisely, we will consider a class of particularly simple categorical models which may not be adequate to model computation in the intuitionistic case, but which look viable for classical LL.

3.1 Copying and discarding data

In a purely linear language, every datum must be used exactly once. For example, a term like $\lambda x.(fx, gx)$ would be illegal because x appears twice in the body of the λ-abstraction. However, even in the pure framework we can actually copy and discard data of certain types without violating linearity. For example, if we define the type *bool* as $1 \oplus 1$, we can in effect duplicate a truth value using the function

$$\lambda x{:}bool.\,\mathbf{if}\ x\ \mathbf{then}\ (f\,\mathrm{true}, g\,\mathrm{true})\ \mathbf{else}\ (f\,\mathrm{false}, g\,\mathrm{false})$$

where true and false abbreviate $\mathbf{inl}\,()$ and $\mathbf{inr}\,()$, respectively, and the **if**-construct denotes the coproduct casing morphism. We note that with this encoding, the function argument will still be evaluated exactly once, regardless of the global evaluation strategy. Of course, an analogous expression can be used to discard a boolean value. But most importantly, in the case where the boolean argument is in fact used exactly once, adding the explicit conditional does not change the meaning of the expression because of the identity

$$\mathbf{if}\ b\ \mathbf{then}\ E\ \mathrm{true}\ \mathbf{else}\ E\ \mathrm{false}\ =\ E\,b$$

(which is just the λ-syntax counterpart of the coproduct uniqueness equation $[f \circ \mathbf{inl}, f \circ \mathbf{inr}] = f$). Thus, the linear case integrates smoothly with the extension.

In general, we can copy/discard finite tensor products (component-wise) and categorical coproducts (case-wise) of already copyable/discardable objects. With recursively defined types, such as $N \cong 1 \oplus N$ and some form of primitive recursion, we can similarly copy and discard natural numbers (if they are not already included as primitive types with associated copy/discard functions), as well as lists, trees, etc. of such types. However, there is no general way to copy even very "small" function spaces such as $bool \multimap bool$ or categorical products like $bool \,\&\, bool$ because their internal structure is inaccessible.

To formalize this distinction, we must first make precise what it means to copy or discard a value. As usual, category theory provides a tool in the form of *comonoids*. The details can be found in [Mac71] or [Laf88a]; here, we will just state the key concepts:

Definition 4 *A* (co-)commutative comonoid *in a symmetric monoidal category $\mathcal{C}$ consists of an object A and two morphisms $d : A \to 1$ and $c : A \to A \otimes A$ satisfying the following equations (written out in applicative notation with $a : A$):*

$$\mathbf{let}\ (a_1,a_2){=}c(a)\ \mathbf{in}\ \ \mathbf{let}\ ()=d(a_2)\ \mathbf{in}\ a_1 \ =\ a$$

$$\mathbf{let}\ (a_1,a_2){=}c(a)\ \mathbf{in}\ \ \mathbf{let}\ (a_{21},a_{22}){=}c(a_2)\ \mathbf{in}\ (a_1,a_{21},a_{22}) \ =\ \mathbf{let}\ (a_1,a_2){=}c(a)\ \mathbf{in}\ \ \mathbf{let}\ (a_{11},a_{12}){=}c(a_1)\ \mathbf{in}\ (a_{11},a_{12},a_2)$$

$$\mathbf{let}\ (a_1,a_2){=}c(a)\ \mathbf{in}\ (a_2,a_1) \ =\ \mathbf{let}\ (a_1,a_2){=}c(a)\ \mathbf{in}\ (a_1,a_2)$$

These equations ensure [Mac71] that for a given comonoid (A,d,c), there exists a unique "structural morphism" (*i.e.*, one built out of c's, d's, and the isomorphisms of definition 1) $A \to A^n (= A \otimes \overset{n}{\cdots} \otimes A)$ for every $n \geq 0$.

A *morphism of comonoids* $f : (A, d_A, c_A) \to (B, d_B, c_B)$ is a $\mathcal{C}$-morphism $f : A \to B$ that respects comonoidal structure, *i.e.*, satisfies

$$\begin{aligned} d_B(f(a)) &= \mathbf{let}\ () = d_A(a)\ \mathbf{in}\ () \\ c_B(f(a)) &= \mathbf{let}\ (a_1,a_2) = c_A(a)\ \mathbf{in}\ (f(a_1), f(a_2)) \end{aligned}$$

Not every value can be copied or discarded, but if it can, it seems natural to require that there be a unique way of doing so. A simple, if not very subtle, way of ensuring this is provided by the following:

Definition 5 *A symmetric monoidal category $\mathcal{C}$ will be called* pre-cartesian *if the forgetful functor from the category of commutative comonoids in $\mathcal{C}$ to $\mathcal{C}$ is a* full embedding, i.e., *if it determines a full subcategory $Core(\mathcal{C})$ of $\mathcal{C}$.*

We will generally use the letters S and T for objects in this subcategory. In effect, the definition associates to every object of the core a unique way to copy and discard values of that type. For every such object S, we thus have a *unique* comonoid $(S, \mathbf{drop}_S : S \to 1, \mathbf{copy}_S : S \to S \otimes S)$, and *all* morphisms between such objects are morphisms of comonoids. Coupled with the properties of the tensor products, this gives us the easily (if somewhat tediously) checked

Proposition 1 *In a pre-cartesian symmetric monoidal category $\mathcal{C}$, the subcategory $Core(\mathcal{C})$ is in fact cartesian, with 1 as a terminal object and the tensor product $S \otimes T$ as a categorical product of S and T in $Core(\mathcal{C})$.*

Motivated by this result, we will refer to objects in the subcategory as *cartesian* objects or types. For such objects we do have the usual projections and pairing associated with categorical products. Moreover, $\mathit{0}$ is cartesian, and if S and T are, so is $S \oplus T$.

Since there is only one way to copy or discard data, we could simply extend the linear λ-syntax to allow multiple occurrences of cartesian-typed variables without semantic ambiguity. However, such a syntax would be slightly deceptive, because conversions such as $(\lambda x.\,(x,x))\,E = (E,E)$ are only valid when all the free variables of E are of cartesian type (the morphism corresponding to E must be a morphism of comonoids to commute with copying). Instead, we will use the notation $\lambda^* x.\,E$ with x occurring multiple times in E as a conceptual abbreviation for the purely linear λ-term with all copying and discarding made explicit as outlined in the beginning of this section. This does not mean that it has to be *implemented* that way, only that its semantic properties are defined by the expansion. In the case where x is actually used exactly once in E, we do have $\lambda^* x.\,E = \lambda x.\,E$.

3.2 The modality "of course"

Armed with a categorical characterization of copying and discarding, let us now consider a "completion" of the framework to arbitrary types, based on the concept of representations in a subcategory. For example, the category of sets and (total) functions is clearly a proper subcategory of sets and binary relations, but every relation $A \to B$ can be *uniquely encoded* as a function $A \to \mathcal{P}(B)$ (*i.e.*, from A to the powerset of B). Analogously, we have:

Definition 6 *A pre-cartesian category $\mathcal{C}$ will be called* canonical *if its core is coreflective in $\mathcal{C}$,* i.e., *if the inclusion functor $i : \mathit{Core}(\mathcal{C}) \hookrightarrow \mathcal{C}$ has a right adjoint $! : \mathcal{C} \to \mathit{Core}(\mathcal{C})$.*

More explicitly, the adjunction assigns to every object A of $\mathcal{C}$ an object $!A$ ("of course A") of $\mathit{Core}(\mathcal{C})$ (*i.e.*, with associated morphisms $\mathbf{drop}_{!A} : !A \to \mathit{1}$ and $\mathbf{copy}_{!A} : !A \to !A \otimes !A$) together with a morphism $\mathbf{read}_A : !A \to A$, such that for every object S of $\mathit{Core}(\mathcal{C})$ and morphism $f : S \to A$, there exists a unique morphism $\mathbf{code}(f) : S \to !A$ satisfying

$$\mathbf{read}_A \circ \mathbf{code}(f) \;=\; f : S \to A$$

Uniqueness is expressed by the matching equation $\mathbf{code}(\mathbf{read}_A \circ g) = g : S \to !A$, which again is equivalent (given the equation above) to the following, more intuitively appealing pair (where $h : T \to S$ is any core morphism):

$$\begin{aligned} \mathbf{code}(\mathbf{read}_A) &= \mathbf{id}_{!A} : !A \to !A \\ \mathbf{code}(f) \circ h &= \mathbf{code}(f \circ h) : T \to !A \end{aligned}$$

As expected from the computational interpretation of the products and "!", we have $!(A \,\&\, B) \cong !A \otimes !B$ and $!\top \cong \mathit{1}$. The quickest (if somewhat obscure) way to verify this is to note that the functor ! has a left adjoint (namely the inclusion), hence preserves limits. In particular, it maps categorical products in the large category to categorical products in the subcategory. Fortunately, we can also easily construct the required isomorphisms explicitly. Moreover, for any object S of the core, there is an isomorphism $!S \cong S$; in particular, $!!A \cong !A$ for every object A, so we do not have to deal with multiple "degrees" of copyability.

A computational interpretation of $!A$ in this framework is as a pair consisting of a (copyable and discardable) value of a *representation type* S and an *access function* (a code pointer, not a closure) of type $S \to A$. We copy or discard a value of type $!A$ by copying or discarding the S-typed value representing it. Of course, the actual representation is completely invisible from "within" the category. The following analogy may be useful: we can *implement* higher-order functions using simpler constructs to build and manipulate explicit closures, but at the price of extensionality: two closures may represent the same function even if their internal structure is completely different. Similarly, the categorical equations governing $!A$ permit us to change the representation of an $!A$-typed value at any time, as long as it **read**s to the same value.

In particular, if A is itself a cartesian type, we can change the representation type to just A and the access function to $\mathbf{id}_A$ whenever convenient – an obvious choice being immediately after the first **read** of $!A$. In other words, the usual technique of *memoization* applies, even in the presence of first-class continuations.

In the linear λ-calculus, we can represent $\mathbf{code}(-)$ as the term $'E : !A$, where all the free variables in $E : A$ have cartesian (but not necessarily "!"-) type. Conversely, we add a new abstraction pattern $'x : !A$, with every use of $x : A$ in the body representing a $\mathbf{read}_A$.[2] Unlike λ^*, which was only a definitional extension, the '-notation represents additional categorical structure; in particular, $\lambda^* x{:}!A.\,E := \lambda' a{:}!A.\,E['a/x]$.

Taken together with the pre-cartesian structure of $\mathcal{C}$, the three adjunction equations for $\mathbf{code}(-)$ and **read** correspond (though not quite 1-1) to the following three equalities between linear λ-terms:

$$\begin{aligned} (\lambda' x.\,E_1)\,'E_2 &= E_1[E_2/x] \\ \lambda' x.\,E\,'x &= E \quad (x \notin FV(E)) \\ (\lambda^* x.\,'E_1)\,E_2 &= '(E_1[E_2/x]) \end{aligned}$$

[2] The syntactic similarity with the Lisp/Scheme `quote` construct is, of course, purely coincidental. We use ' instead of ! on purpose; to maintain a simple correspondence between λ-terms of functional type and categorical morphisms, the operator ! must be reserved for functorial action on morphisms: $!f = \lambda' x.\,'(fx)$.

The first two of these allow us to extend "pattern-$\beta\eta$" equivalence to '-patterns. The third verifies correctness of substitution under '; we need the λ^* here, because even if x only *occurs* once in E_1, it will be embedded in a value that can be *used* multiple times.

Again, it is worth pointing out that though the above discussion does not explicitly mention classical features of the category, the whole model is only reasonable because we still have an ace left for expressing potentially non-terminating computations: discardable continuations.

3.3 The modality "why not"

Finally, let us consider how we can introduce partiality in a linear category:

Definition 7 *We write* $?A$ *("why not* A*") for* $(!A^\perp)^\perp$.

Computationally, we interpret the isomorphism $A^{\perp\perp} \cong A$ as "a continuation t accepting an A-accepting continuation k must eventually apply k to some A-typed value a." But $?A$ represents a continuation t that accepts a continuation k which it may freely discard or copy. Thus, evaluation of a $?A$-typed term may never return, or it may return multiple times to the same point with different values.

At least the non-returning aspect is common to all programming languages with the possibility of infinite loops, even "intuitionistic linear" ones like Lafont's LIVE [Laf88b]: a linear function which "loops forever" has in a sense broken its promise to apply the return continuation it is passed. We can say that such functions are not linear in their *output*, even if their *input* is only used once. On the other hand, a type system based on classical linear logic allows us to express that a function always *produces* exactly one result (essentially a *totality* condition) using the same mechanisms as for expressing that the input is *consumed* exactly once (a refinement of *strictness*).

4 Representing Evaluation Strategies

This section sketches an application of classical linear logic as a semantic basis for traditional functional languages. In particular, we focus on how compound data types and evaluation strategies are represented by constructs in the "subcategory" models of classical linear logic presented in the previous section. We will consider call-by-value (CBV) and call-by-name (CBN) evaluation of a simple language with product, coproduct, exponential and "co-exponential" types (the last-mentioned seem to arise naturally in categorical models of first-class continuations [Fil89a]). In a linear framework, these two strategies can be seen as essentially *dual*, a relation usually obscured by the asymmetries of intuitionistic logic.

4.1 Call-by-value

A fairly convincing case can be made that a CBV functional language should have some representation of first-class continuations for completeness, much as a product type falls out naturally from the correspondence between ordinary lambda-calculi and CCCs. Modulo syntactic differences, languages of this kind include (the functional subsets of) Scheme [RC86] and recent versions of Standard ML of New Jersey [DHM91].

Since CBV continuations are first-class (hence copyable and discardable), it is convenient to introduce the following abbreviation for *call-by-value negation*:

$$\neg A \;:=\; !A^\perp$$

We note in particular that $\neg \mathit{0} \cong \mathit{1}$ and $\neg(A \oplus B) \cong \neg A \otimes \neg B$, *i.e.*, a CBV continuation expecting a coproduct-typed value is equivalent to a pair of continuations, one for each case. Building on this concept of negation, we interpret a CBV term E with a free variable x as a linear morphism $[\![x{:}A \vdash E{:}B]\!] : [\![A]\!] \to ?[\![B]\!]$ (or, equivalently: $[\![A]\!] \otimes \neg[\![B]\!] \to \perp$), where the meaning of CBV types is given by:

$$\begin{array}{rcll}
[\![P]\!] & = & !P & (P \text{ a base type}) \\
[\![\mathbf{1}]\!] & = & \mathit{1} & \\
[\![A \times B]\!] & = & [\![A]\!] \otimes [\![B]\!] & \\
[\![\mathbf{0}]\!] & = & \mathit{0} & \\
[\![A + B]\!] & = & [\![A]\!] \oplus [\![B]\!] & \\
[\![[A \to B]]\!] & = & \neg([\![A]\!] \otimes \neg[\![B]\!]) \;\cong\; !([\![A]\!] \multimap ?[\![B]\!]) & \\
[\![[A \leftarrow B]]\!] & = & \neg[\![A]\!] \otimes [\![B]\!] \;\cong\; !([\![A]\!] \multimap \perp) \otimes [\![B]\!] &
\end{array}$$

($[A \leftarrow B]$ is a coexponential type, see below.) For example, we find $[\![[A \to \mathbf{0}]]\!] \cong \neg[\![A]\!]$ and $[\![[\mathbf{1} \to A]]\!] \cong \neg\neg[\![A]\!] \cong !?[\![A]\!]$. We note that all "expressible" types are cartesian (in particular, for any type T of the CBV language, $[\![T]\!] \cong ![\![T]\!]$). This means that a CBV term can have multiple free variables, each of which it may use any number of times, as usual.

The actual translation on CBV λ-terms is essentially a typed and uncurried version the well-known continuation-passing transform (*e.g.*, [Plo75]), but the key observation here is that the resulting terms are *linear*! In particular, we can simply use λ^*-s in the translation to express multiple uses of data in the original terms because all the values involved are cartesian ($\perp$, representing the type of final answers, is not cartesian but never appears as a function argument either).

This suggests that continuation-passing style is much more than an ad-hoc syntactic restriction on non-linear terms; rather, it is in a sense the "real meaning" of CBV

terms, and thus a very natural intermediate representation for compilation [Ste78, AJ89], semantic analysis [Shi91], or partial evaluation [CD91]. The linear framework can easily encompass *trivial functions* (*i.e.*, functions that always terminate and do not escape) as values of type $\neg(A \otimes B^\perp)$. Also, CBV continuations are necessarily *strict* by virtue of their types.

Furthermore, we observe that $\mathbf{0}$ is in fact a categorical initial object in the category of CBV types and terms, and $A+B$ is a categorical coproduct. We even have *cocartesian closure*: the coproduct has a left adjoint (the coexponential) and functions that return a coproduct-typed result can be *co-curried* by distinguishing one of the cases as the "explicit return" and the other as an "implicit return" corresponding to a non-local exit.

On the other hand, the type denoted by $A \times B$ is not a categorical product: $\mathbf{fst}\,(E_1, E_2) \neq E_1$ when E_2 is not total. We can, however, define an alternative *lazy product* within the language:

$$A \amalg B \;:=\; [\mathbf{1} \to A] \times [\mathbf{1} \to B]$$

with interpretation

$$[\![A \amalg B]\!] \;\cong\; !?[\![A]\!] \otimes !?[\![B]\!] \;\cong\; !(?[\![A]\!] \,\&\, ?[\![B]\!])$$

This is not a categorical product either (we have existence but not uniqueness of mediating morphisms), but it has an interesting parallel in CBN coproducts, as we will see in the next subsection.

4.2 Call-by-name

Similarly to CBV, we will consider models of "full CBN" languages; the properties of coexponential types in CBN turn out not to interfere significantly with the rest of the language, but still provide a notion of first-class continuations appropriate to this strategy.

Let us first consider the "pure CBN" language [Fil89b], obtained as the exact mirror image of CBV. Here, an expression $E : B$ with free variable $x : A$ is represented by a linear morphism $![\![A]\!] \to [\![B]\!]$ (or $![\![A]\!] \otimes [\![B]\!]^\perp \to \perp$). Thus, CBN terms can also use their free variables any number of times, but for a slightly different reason than in CBV. We also have a *CBN negation*:

$$\sim A \;:=\; ?\,A^\perp$$

and the following interpretation of types:

$$\begin{array}{rcl}
[\![P]\!] & = & ?P \qquad (P \text{ a base type}) \\
[\![\mathbf{1}]\!] & = & \top \\
[\![A \times B]\!] & = & [\![A]\!] \,\&\, [\![B]\!] \\
[\![\mathbf{0}]\!] & = & \perp \\
[\![A + B]\!] & = & [\![A]\!] \parr [\![B]\!] \\
[\![[A \to B]]\!] & = & \sim[\![A]\!] \parr [\![B]\!] \;\cong\; ![\![A]\!] \multimap [\![B]\!] \\
[\![[A \leftarrow B]]\!] & = & \sim([\![A]\!] \parr \sim[\![B]\!]) \;\cong\; ?([\![A]\!]^\perp \otimes ![\![B]\!])
\end{array}$$

Following Girard's notation, we use $A \parr B$ to denote $(A^\perp \otimes B^\perp)^\perp$. We note that all types are now cocartesian, *i.e.*, for any type expression T, $[\![T]\!] \cong ?[\![T]\!]$. This means that all variables represent potentially nonterminating residual computations. Of course, since the connectives used for CBN can be expressed (but at the cost of obscuring the symmetry) in terms of the CBV connectives and linear negation, the above translation on types can likewise be used as the basis of a CPS-based implementation.

It is easily checked that the CBN category is in fact cartesian closed, with $\mathbf{1}$ as terminal object, $A \times B$ as product and $[A \to B]$ as exponential. However, the behavior of $A+B$ is not what we would expect – it is "too lazy." In fact, there is no way to add proper coproducts to a language where every function has a fixpoint [HP90], but just as for lazy products in CBV, we can define the more conventional *eager coproducts*

$$A \amalg B \;:=\; [\mathbf{0} \leftarrow A] + [\mathbf{0} \leftarrow B]$$

with the interpretation

$$[\![A \amalg B]\!] \;\cong\; ?![\![A]\!] \parr ?![\![B]\!] \;\cong\; ?(![\![A]\!] \oplus ![\![B]\!])$$

which make it possible to evaluate a coproduct-typed datum until its injection tag is known but without forcing evaluation of the actual inject. Except for the ? (which accounts for possible non-termination), this is identical to the usual encoding of intuitionistic "or" in LL [GL87]. In particular, we have

$$[\![\mathit{bool}]\!] \;=\; ?\mathit{bool} \;=\; ?(\mathit{1} \oplus \mathit{1}) \;\cong\; ?(!\top \oplus !\top) \;\cong\; [\![\mathbf{1} \amalg \mathbf{1}]\!]$$

so booleans are definable as expected.

5 Comparison with Related Work

Naturally, this paper builds on Girard's fundamental work on Linear Logic [Gir87]. The question/answer symmetry mentioned there is interpreted here as a duality between values and continuations, seen as the two extreme cases of linear functions.

The interpretation of modality-free classical LL as a linear category [See89, MOM90] appears to be commonly accepted. However, the proposed categorical view of "!"-types as cofree coalgebras [GL87] seems oriented towards the intuitionistic subset of LL only. The additional structure of classical LL, notably the availability of "?"-types for representing terms that do not necessarily evaluate to proper values should allow for conceptually, as well as computationally, simpler models.

Both categorical and more pragmatic computational interpretations of linear logic as a *functional* program-

ming language, *e.g.*, [Hol88] have focused on the intuitionistic subset, with the understanding that the classical version was intimately linked with parallelism and communication; more recent work [Abr90] reinforces this dichotomy. However, while its potential for parallel evaluation is indeed exciting, it seems that classical LL can also give us a better understanding of control flow in traditional, sequential languages than either intuitionistic or intuitionistic linear logic.

From another direction, syntactic theories such as the "continuation calculi" of [FFKD87] have been proposed as a tool for formal reasoning about `call/cc`-like control operators. The striking connection between such operators and classical logic was later pointed out by Griffin [Gri90], and very convincingly related to Friedman's A-translation and CPS transformations by Murthy [Mur91]. There is hope that an analysis in terms of (classical) linearity will also make it possible to derive such results *semantically*, supplementing the current understanding of CPS-translation as an essentially *syntactic* concept.

A recent paper [Nis91] notes a connection between classical LL and Scheme-like languages, and presents a somewhat involved translation from typed λ-terms with *call/cc* to Girard's "proof nets" for LL. It too, however, takes a strongly syntax-based approach and does not consider the semantic implications of linear control. (In fact, the translation looks conceptually similar to simply a Griffin-Murthy CBV CPS-translation from classical to intuitionistic logic, followed by Girard's translation from intuitionistic to linear logic).

Finally, some work independently motivated by the symmetry between values and continuations is reported in [Fil89a, Fil89b]. In retrospect, the kind of duality considered there mirrors the CBV/CBN negations of section 4, rather than the underlying "pure" linear negation, leading to a somewhat ad-hoc axiomatization. Nevertheless, the results obtained there formed the principal motivation for the present investigation.

6 Conclusion and Issues

We have presented a view of linear negation as representing first-class continuations in linear lambda-calculi. In particular, the dualizing construct in categorical models of classical linear logic can be interpreted as a `call/cc`-like control operator. Building on this, we have shown how the modality "!", when applied to continuation types, lets us delineate the class of possibly non-returning computations as those which potentially discard their return continuations. This allows us to decouple delayed evaluation from "!"-types and leads to a simpler "coreflective subcategory" interpretation of the latter. Finally, we have sketched how the additional expressive power of classical linear logic allows us to accurately represent data types and control flow in traditional functional programming languages.

Clearly, much work remains in formalizing what has been outlined in this paper. However, a full understanding of linear continuations will probably also involve resolution of two closely related issues:

- **Polymorphism.** The striking similarity between the isomorphisms $(A \multimap \bot) \multimap \bot \cong A$ in linear categories and $\forall C.(A \to C) \to C \cong A$ in parametric models of the second-order polymorphic λ-calculus [Rey83] seems to be more than a coincidence; a formal connection would be an important result.

- **The general CPS transform.** By changing the codomain of continuations from $\bot$ to an information-carrying type, it seems possible to represent a wide variety of additional computational structure, such as side effects and backtracking, in a way complementary to the computational monads of [Mog89]. Some preliminary results in this direction are reported in [DF90], but the topic is far from explored.

Acknowledgments

I wish to thank Olivier Danvy, Timothy Griffin, Robert Harper, Narciso Martí-Oliet, Chetan Murthy, Benjamin Pierce, and John Reynolds for their insightful comments and helpful suggestions on various drafts of this paper.

References

[Abr90] Samson Abramsky. Computational interpretations of linear logic. Imperial College Research Report DOC 90/20, Department of Computing, Imperial College of Science, Technology and Medicine, London, UK, 1990.

[AJ89] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, January 1989.

[Bar79] Michael Barr. **-Autonomous Categories*. Number 752 in Lecture Notes in Mathematics. Springer-Verlag, 1979.

[CD91] Charles Consel and Olivier Danvy. For a better support of static data flow. In *Proceedings of the 1991 Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 496–519, Cambridge, Massachusetts, August 1991.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990.

[DHM91] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Orlando, Florida, January 1991.

[FFKD87] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[Fil89a] Andrzej Filinski. Declarative continuations: An investigation of duality in programming language semantics. In David H. Pitt et al., editors, *Category Theory and Computer Science*, number 389 in Lecture Notes in Computer Science, pages 224–249, Manchester, UK, September 1989.

[Fil89b] Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, Computer Science Department, University of Copenhagen, August 1989. DIKU Report 89/11.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GL87] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In *Proceedings of TAPSOFT '87*, number 250 in Lecture Notes in Computer Science, pages 52–66, Pisa, Italy, March 1987.

[Gri90] Timothy G. Griffin. A formulae-as-types notion of control. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990.

[Hol88] Sören Holmström. A linear functional language. In Thomas Johnsson, Simon Peyton Jones, and Kent Karlsson, editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pages 13–32, Aspenäs, Sweden, September 1988. Chalmers University PMG Report 53.

[HP90] Hagen Huwig and Axel Poigné. A note on inconsistencies caused by fixpoints in a cartesian closed category. *Theoretical Computer Science*, 73:101–112, 1990.

[Laf88a] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[Laf88b] Yves Lafont. *Logiques, Catégories et Machines*. PhD thesis, Université de Paris VII, Paris, France, January 1988.

[LS86] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[Mac71] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE.

[MOM90] Narciso Martí-Oliet and José Meseguer. Duality in closed and linear categories. Technical Report SRI-CSL-90-01, SRI International, Menlo Park, California, February 1990.

[MOM91] Narciso Martí-Oliet and José Meseguer. From Petri nets to linear logic. *Mathematical Structures in Computer Science*, 1, 1991.

[Mur91] Chetan R. Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, Amsterdam, The Netherlands, July 1991.

[Nis91] Shin-ya Nishizaki. Programs with continuations and linear logic. In *International Conference on Theoretical Aspects of Computer Science*, pages 513–531, Sendai, Japan, September 1991.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[RC86] Jonathan Rees and William Clinger (editors). Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. IFIP, 1983.

[See89] Robert A. G. Seely. Linear logic, *-autonomous categories and cofree coalgebras. In John W. Gray and Andre Scedrov, editors, *Proceedings of the AMS-IMS-SIAM Joint Conference on Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 371–382, Boulder, Colorado, 1989. American Mathematical Society.

[Shi91] Olin Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–198, New Haven, Connecticut, June 1991. SIGPLAN Notices, Vol. 26, No. 9.

[Ste78] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artficial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[Wad90] Philip Wadler. Linear types can change the world! In *Proceedings of IFIP TC2 Working Conference on Programming Concepts and Methods*, pages 546–566, Sea Galilee, Israel, April 1990.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 LISP Conference*, pages 19–28, Stanford, California, August 1980.

A A quick review of some categorical terminology

Please note: this appendix contains no original material and is included only to help readers unfamiliar with category theory follow the technical contents of the paper. The definitions presented here are necessarily terse and not completely rigorous; the interested reader should consult a proper introduction to category theory for the full story.

A *category* consists of a collection of *objects* and *morphisms* (or *arrows*) between them, such that (1) any pair of morphisms $f : A \to B$ and $g : B \to C$ can be *composed* into $g \circ f : A \to C$, (2) morphism composition is associative ($(h \circ g) \circ f = h \circ (g \circ f)$), and (3) for every object A, there exists an *identity* morphism $\mathbf{id}_A : A \to A$ such that for $f : A \to B$, $\mathbf{id}_B \circ f = f = f \circ \mathbf{id}_A$. An important class of such structures are the *concrete* categories, where objects are sets with some structure and morphisms are structure-preserving maps. Examples include the categories **Set** of sets and (total) functions, $\mathbf{Set}_p$ of sets and partial functions, **Grp** of groups and group homomorphisms, **Dom** of (Scott) domains and continuous functions, *etc.* Similarly, the types of a functional language L (*e.g.*, ML) and computable functions between them form a category $\mathbf{Typ}_L$.

An *isomorphism* is a morphism $f : A \to B$ with a two-sided inverse $f^{-1} : B \to A$ (*i.e.*, $f^{-1} \circ f = \mathbf{id}_A$ and $f \circ f^{-1} = \mathbf{id}_B$); in this case we say that A and B are *isomorphic*, written $A \cong B$. In most contexts, isomorphic objects can be treated as identical because they have the same categorical properties. A *subcategory* $\mathcal{S}$ of a category $\mathcal{C}$ is a subset of $\mathcal{C}$'s objects and morphisms which is itself a category; for example, **Set** is a subcategory of $\mathbf{Set}_p$. A *full* subcategory contains all morphisms of the original category whose source and target objects are in the subcategory; the category $\mathbf{Set}_f$ of *finite* sets and (total) functions between them is a full subcategory of **Set**.

A key observation of category theory is that many important concepts can be defined without ever referring to he internal structure of objects, only to the morphisms between them. For example, a *categorical product* of two objects A and B is given by an object $A \times B$ together with *projections* $\mathbf{fst} : A \times B \to A$ and $\mathbf{snd} : A \times B \to B$, such that for any object C and morphisms $f : C \to A$ and $g : C \to B$, there exists a unique *pairing* morphism $\langle f, g \rangle : C \to A \times B$ satisfying $\mathbf{fst} \circ \langle f, g \rangle = f$ and $\mathbf{snd} \circ \langle f, g \rangle = g$. The uniqueness condition can be expressed equationally as $\langle \mathbf{fst} \circ h, \mathbf{snd} \circ h \rangle = h$ for all $h : C \to A \times B$. A *terminal* object $\mathbf{1}$ has the property that for any object A, there exists a unique morphism $\langle\,\rangle_A : A \to \mathbf{1}$. We can think of a terminal object as a nullary product; in particular, $A \times \mathbf{1} \cong A$ for any object A. In **Set**, the categorical product of two objects is actually their cartesian product with the obvious projections and pairing; any one-element set is terminal.

As often happens, we can *dualize* these definitions by reversing the direction of all arrows. Thus, a *coproduct* of A and B consists of an object $A+B$ and two *injections* $\mathbf{inl} : A \to A + B$ and $\mathbf{inr} : B \to A + B$, such that for any object C and morphisms $f : A \to C$ and $g : B \to C$ there exists a unique "*casing*" morphism $[f, g] : A+B \to C$ satisfying $[f, g] \circ \mathbf{inl} = f$ and $[f, g] \circ \mathbf{inr} = g$. Again, we can express uniqueness as the equation $[h \circ \mathbf{inl}, h \circ \mathbf{inr}] = h$ for all $h : A + B \to C$. An object is *initial* if for every object A, there exists a unique morphism $[]_A : \mathbf{0} \to A$; an initial object is a degenerate coproduct: $A + \mathbf{0} \cong A$. In **Set**, the disjoint (*i.e.*, *tagged*) union of two sets is their coproduct; the empty set is the only initial object.

Category theory also provides a natural abstraction of higher-order functions: an *exponential* of two objects B and C consists of an object $B \Rightarrow C$ and an *application* morphism $\mathbf{ap} : (B \Rightarrow C) \times B \to C$ such that any morphism $f : A \times B \to C$ has a unique *curried form* $\mathbf{cur}(f) : A \to B \Rightarrow C$ with the property that $\mathbf{ap} \circ \langle \mathbf{cur}(f) \circ \mathbf{fst}, \mathbf{snd} \rangle = f$. Here, the uniqueness equation is $\mathbf{cur}(\mathbf{ap} \circ \langle g \circ \mathbf{fst}, \mathbf{snd} \rangle) = g$ for all $g : A \to B \Rightarrow C$. In **Set**, $A \Rightarrow B$ is the function space B^A together with the expected interpretations of application and currying. A category which has all finite products is called *cartesian*. If it also has all exponentials, it is called *cartesian closed*, commonly abbreviated to *CCC*.

A *functor* $F : \mathcal{C} \to \mathcal{D}$ from a category $\mathcal{C}$ to a category $\mathcal{D}$ is a mapping from $\mathcal{C}$-objects A to $\mathcal{D}$-objects $F(A)$ and $\mathcal{C}$-morphisms $f : A \to B$ to $\mathcal{D}$-morphisms $F(f) : F(A) \to F(B)$, respecting identity and composition (*i.e.*, $F(\mathbf{id}_A) = \mathbf{id}_{F(A)}$ and $F(f \circ g) = F(f) \circ F(g)$).

For example, the functor $list : \mathbf{Typ} \to \mathbf{Typ}$ maps a type A to the type $list(A)$ and a function $f : A \to B$ to the function $maplist(f) : list(A) \to list(B)$. The definition generalizes easily to functors of several arguments; in particular, a functor of two variables is often called a *bifunctor*. The binary products and coproducts defined above can be viewed as bifunctors if we define their actions on morphisms as $f \times g = \langle f \circ \mathbf{fst}, g \circ \mathbf{snd} \rangle$, and $f + g = [\mathbf{inl} \circ f, \mathbf{inr} \circ g]$. The exponential is also a bifunctor, but is *contravariant* (reverses arrow direction) in its first argument.

A natural transformation $\eta : F \dot{\to} G$ between two functors $F, G : \mathcal{C} \to \mathcal{D}$ is a collection, indexed by $\mathcal{C}$-objects X, of $\mathcal{D}$-morphisms $\eta_X : F(X) \to G(X)$ such that for any $\mathcal{C}$-morphism $f : A \to B$, $\eta_B \circ F(f) = G(f) \circ \eta_A : F(A) \to G(B)$. To a first approximation, we can think of natural transformations in a programming language setting as polymorphic functions. For example, consider the type-indexed collection of functions $flatten_A : tree(A) \to list(A)$. If $zerop : int \to bool$, is a function, naturality ensures that $flatten_{bool} \circ maptree(zerop) = maplist(zerop) \circ flatten_{int} : tree(int) \to list(bool)$. (Unfortunately, this analogy does not extend directly to higher-order functions.) A *natural isomorphism* is a natural transformation, all of whose components are isomorphisms.

Finally, a functor $F : \mathcal{C} \to \mathcal{D}$ has a *right adjoint* $G : \mathcal{D} \to \mathcal{C}$ (or, equivalently, G has a *left adjoint* F) if there exists a natural transformation $\varepsilon_X : F(G(X)) \to X$ (the *counit* of the adjunction) such that for every $\mathcal{D}$-morphism $f : F(A) \to B$, there exists a unique $\mathcal{C}$-morphism $f^* : A \to G(B)$ satisfying $\varepsilon_B \circ F(f^*) = f$. For example, for a given object C, the functor $F(X) = X \times C$ has the right adjoint $G(Y) = C \Rightarrow Y$. ε_B is the morphism $\mathbf{ap} : (C \Rightarrow B) \times C \to B$, and for any function $f : A \times C \to B$, there exists a unique morphism $f^* = \mathbf{cur}(f) : A \to C \Rightarrow B$ satisfying $\mathbf{ap} \circ (f^* \times \mathbf{id}_C) = f$. Many of the other categorical definitions above (products, coproducts, initial and terminal objects) are also examples of adjoint situations.