A Cartesian Closed Category of Parallel Algorithms between Scott Domains

Stephen Brookes Shai Geva

July 1991
CMU-CS-91-159 2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to Proceedings of Dagstuhl Seminar on
Programming Language Semantics and Model Theory,
Dagstuhl Castle, June 1991.

This research was supported in part by National Science Foundation grant CCR-9006064 and in part by DARPA/NSF grant CCR-8906483.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

Keywords: theory, applicative (functional) programming, semantics, parallelism, category theory

Abstract

We present a category-theoretic framework for providing intensional semantics of programming languages and establishing connections between semantics given at different levels of intensional detail. We use a comonad to model an abstract notion of computation, and we obtain an intensional category from an extensional category by the co-Kleisli construction; thus, while an extensional morphism can be viewed as a function from values to values, an intensional morphism is akin to a function from computations to values. We state a simple category-theoretic result about cartesian closure. We then explore the particular example obtained by taking the extensional category to be **Cont**, the category of Scott domains with continuous functions as morphisms, with a computation represented as a non-decreasing sequence of values. We refer to morphisms in the resulting intensional category as algorithms. We show that the category **Alg** of Scott domains with algorithms as morphisms is cartesian closed. We define an intensional partial order on algorithms, with respect to which application, currying and composition are continuous. We show that every algorithm determines a continuous input-output function, and that every continuous function is the input-output function of some algorithm. This is in contrast to the sequential algorithms model of Berry and Curien, in which algorithms determine the sequential input-output functions (between concrete domains). Since the continuous functions include inherently non-sequential functions such as parallel-or, we designate our algorithms as parallel. Two algorithms are input-output equivalent iff they have the same input-output function. The intensional ordering on algorithms collapses, under input-output equivalence, onto the pointwise ordering on their input-output functions. We define an intensional semantics of the simply typed λ-calculus, and relate it to the standard extensional semantics. We discuss related work and we propose some topics for further research.

1 Introduction

Most work on denotational semantics of programming languages has focussed on extensional aspects of program behavior, such as the partial function from states to states computed by an imperative program, or the continuous function denoted by a term of the λ-calculus. It is difficult to use such a semantics to reason about intensional properties of programs (such as complexity), since such semantic models usually ignore the computation strategy of a program and therefore give the same meaning to all programs for the same function (for instance, all sorting programs have the same extensional meaning).

A notable exception is the work of Berry and Curien [Cur86] on sequential algorithms, building on previous work of Kahn and Plotkin [KP78] on sequential functions and concrete data structures. This work gave an elegant semantical account of sequential computation on concrete data structures. It is based on a coroutine-like operational semantics [KM77] with a lazy evaluation mechanism. Intuitively, a sequential algorithm reacts to a request for computation of an output value (in a specified output cell) by issuing a sequence of requests for evaluation of input cells; once this sequential computation has been completed an output value may be produced. It is obvious why such operational behavior is sequential, and there is an appropriate connection with the domain-theoretic notion of sequential function as given by Kahn and Plotkin: a Berry-Curien sequential algorithm may be viewed as a sequential function paired with a (sequential) computation strategy. It is therefore possible to use sequential algorithms as intensional meanings of sequential programs, and to make distinctions among sequential algorithms for the same function, based on their computation strategies.

Our general aim is to develop a theory of intensional semantics with appealing mathematical and categorical properties, which supports reasoning about both extensional and intensional properties of programs. We are particularly interested in generalizing the Berry-Curien notion of algorithm to obtain an intensional model of parallel computation. Again, we take an underlying operational semantics based on lazy evaluation. However, in contrast to the sequential case, we allow a parallel algorithm to respond to an output request by starting several input evaluations in parallel and specifying several alternative resumption conditions, in terms of the results of some or all of these input evaluations. This obviously generalizes the sequential interpretation described above, in a simple and natural manner. Our thesis is that parallel algorithms ought to correspond to *continuous* functions paired with (parallel) computation strategies. As evidence in support of this thesis we note that Plotkin [Plo77] showed that the continuous functions model of the simply typed λ-calculus is inherently parallel, since it contains non-sequential functions like parallel-or. A touchstone for judging the viability of our approach is to show that (like Berry and Curien) we obtain a cartesian closed category.

In an earlier paper [BG90] we introduced the query model of parallel algorithms between concrete data structures, and defined application and currying on these algorithms. We introduced an intensional ordering on algorithms, and showed that application and currying are continuous operations with respect to this ordering. The intensional order relates two algorithms if they compute functions related under the pointwise (Scott) ordering, with suitably related computation strategies. However, the query model is only adequate at first-order types, since the structure of queries is insufficiently detailed to provide a proper account of the behavior of higher-order algorithms. Moreover, we were not able to formulate an appropriate notion of composition for algorithms expressed in the query model. The difficulties were caused partly by some technical details in our model construction and partly by our presentation of algorithms in the setting of concrete data structures.

In this paper we move away from concrete data structures and their order-theoretic counterparts, concrete domains, since our concern is with parallelism rather than sequentiality and we no longer need the "concrete" structure (the notion of "cells"). Instead we will work with Scott domains [Sco82]. We develop a categorical semantics of algorithms, where an algorithm is regarded as a continuous function from computations to values, and where the notion of computation is suitably formalized. We define the input-output function of a parallel algorithm, and show that every continuous function is the input-output function of some algorithm. One may therefore view an algorithm equivalently as a continuous function from values to values, paired with a computation strategy. Although we do not give a formal definition of the notion of computation strategy, we supply some operational intuition to convey the general idea and we discuss a variety of example algorithms to illustrate these points.

The categorical treatment supplied in the first part of the paper is rather general, parameterized by the choice of an underlying category that supplies an "extensional" framework and a comonad that embodies an abstract notion of intensional behavior. Essentially, we develop our categorical semantics of algorithms based on comonads, just as Moggi developed a categorical semantics (with different motivations) based on monads [Mog89]. Comonads and the co-Kleisli category [ML71] neatly embody our abstract view of algorithms. If $\mathcal{C}$ is a cartesian closed category and the comonad T over $\mathcal{C}$ preserves products, then the co-Kleisli category $\mathcal{C}_T$ is cartesian closed [See89].

We then move to the particular cartesian closed category **Cont** of Scott domains and continuous functions, and we define a comonad P that maps a domain D to a domain of "paths" over D, ordered componentwise. Intuitively, a path represents a sequence of computation steps and the ordering measures eagerness. The comonad P preserves finite products. We define an algorithm from D to D' to be a continuous function from PD to D', just a morphism in the co-Kleisli category $\mathbf{Cont}_P$. This category, which we call **Alg**, is cartesian closed; its objects are again Scott domains, but the morphisms are algorithms rather than functions. Every algorithm determines a continuous input-output function, and every continuous function is the input-output function of some algorithm (generally, not unique). In fact, the algorithms for a given continuous function, ordered pointwise, form a complete lattice. Thus, the pointwise ordering on algorithms can be regarded as an intensional ordering, since it permits distinctions and comparisons to be made even among algorithms for the same function. The intensional ordering on algorithms relates properly to the pointwise ordering on continuous input-output functions, in the same sense that the sequential algorithms of Berry and Curien, ordered by set inclusion, relate to the stable ordering on sequential input-output functions.

We define an intensional semantics for the simply typed λ-calculus, and we show that it relates properly to the standard continuous functions semantics: we establish a correspondence for each term M between the intensional meaning of M and the extensional meaning of M. In showing this we make use of logical relations [Sta85].

2 Computations, Comonads and Algorithms

We first present some relevant category theoretic results. Although in the rest of the paper we will be mainly interested in a particular application, we present the background in a rather general way, so that the underlying assumptions can be clearly seen. Because of this generality, our later development can be adapted to build algorithms based on other suitably structured notions of computation.

The basic idea is that, given a category $\mathcal{C}$, we model a notion of computation over $\mathcal{C}$ as a comonad over $\mathcal{C}$, the functor part of which maps an object A to an object TA representing computations over

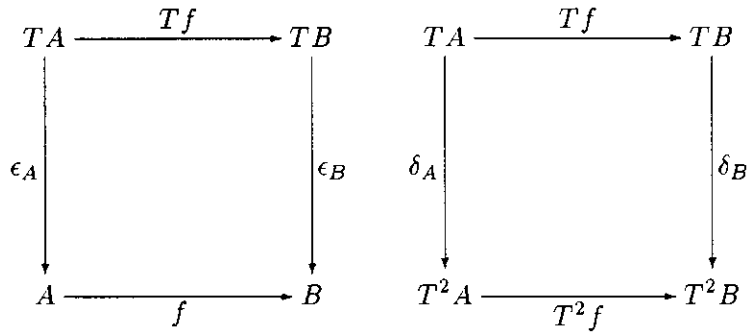

Figure 1: Naturality of ϵ and δ in a comonad: these diagrams commute, for all A, B, $f : A \to^{\mathcal{C}} B$.

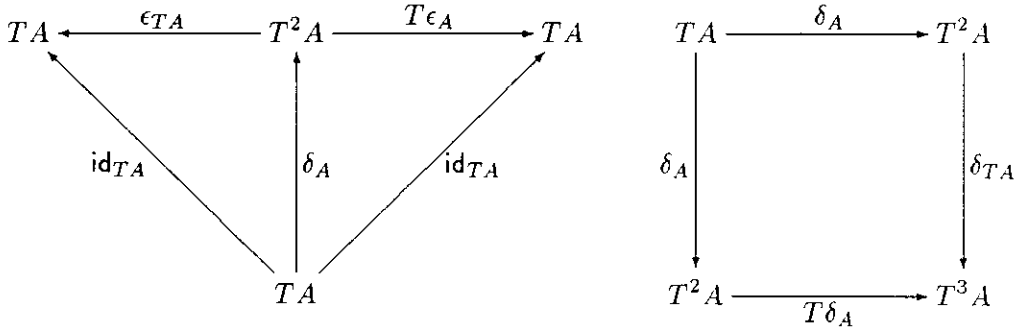

Figure 2: Identity and associativity laws of a comonad: these diagrams commute, for all A.

A. The two other components of the comonad describe how to extract a value from a computation, and how a computation is built up from its sub-computations. We then take an *algorithm* from A to B to be a morphism from TA to B, essentially a function from *input computations* over A to *output values* in B. This leads us to use the co-Kleisli category $\mathcal{C}_T$ [ML71], which has the same objects as $\mathcal{C}$ and in which the morphisms from A to B are exactly the $\mathcal{C}$-morphisms from TA to B.

2.1 Comonads and the co-Kleisli category

Definition 2.1 A *comonad* over a category $\mathcal{C}$ is a triple (T, ϵ, δ) where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\epsilon : T \dot{\to} I_{\mathcal{C}}$ is a natural transformation from T to the identity functor, and $\delta : T \dot{\to} T^2$ is a natural transformation from T to T^2, such that the following associativity and identity conditions hold, for every object A:

$$\begin{aligned} T(\delta_A) \circ \delta_A &= \delta_{TA} \circ \delta_A \\ \epsilon_{TA} \circ \delta_A &= T(\epsilon_A) \circ \delta_A = \mathsf{id}_{TA}. \end{aligned}$$

Figures 1 and 2 express these requirements in diagrammatic form. •

Definition 2.2 Given a comonad (T, ϵ, δ) over $\mathcal{C}$, the *co-Kleisli category* $\mathcal{C}_T$ is defined by:

- The objects of $\mathcal{C}_T$ are the objects of $\mathcal{C}$.
- The morphisms from A to B in $\mathcal{C}_T$ are the morphisms from TA to B in $\mathcal{C}$.

- The identity morphism $\widehat{\mathsf{id}}_A$ on A in $\mathcal{C}_T$ is $\epsilon_A : TA \to^{\mathcal{C}} A$.
- The composition in $\mathcal{C}_T$ of $a : A \to^{\mathcal{C}_T} B$ and $a' : B \to^{\mathcal{C}_T} C$, denoted $a' \bar{\circ} a$, is the composition in $\mathcal{C}$ of $\delta_A : TA \to^{\mathcal{C}} T^2A$, $Ta : T^2A \to^{\mathcal{C}} TB$ and $a' : TB \to^{\mathcal{C}} C$, *i.e.*,

$$a' \bar{\circ} a = a' \circ Ta \circ \delta_A.$$

The associativity and identity laws of the comonad ensure that $\mathcal{C}_T$ is a category [ML71]. •

2.2 Relating $\mathcal{C}$ and $\mathcal{C}_T$

There is an adjoint pair of functors (F, G) between $\mathcal{C}_T$ and $\mathcal{C}$, with the following definitions and properties [ML71]:

- $F : \mathcal{C}_T \to \mathcal{C}$ applies T to objects, $FA = TA$, and for any $a : A \to^{\mathcal{C}_T} B$, $Fa = Ta \circ \delta_A$.
- $G : \mathcal{C} \to \mathcal{C}_T$ is the identity on objects, $GA = A$, and for any $f : A \to^{\mathcal{C}} B$, $Gf = f \circ \epsilon_A = \epsilon_B \circ Tf$ (by naturality of ϵ).
- $T = F \circ G$. The natural transformation $\epsilon : T \dot{\to} I_{\mathcal{C}}$ is the co-unit of the adjunction, and it asserts, together with the unit, a bijection between $TA \to^{\mathcal{C}} B$ and $A \to^{\mathcal{C}_T} B$.

Note also the following identities: $a' \bar{\circ} a = a' \circ Ta \circ \delta_A = a' \circ Fa$, and $\widehat{\mathsf{id}}_A = \epsilon_A = \mathsf{id}_A \circ \epsilon_A = G\,\mathsf{id}$.

The functor G provides a canonical intensional morphism from A to B for every extensional morphism from A to B. The functor F attempts to do the converse, but it "lifts" the type, so that we get an extensional morphism from TA to TB from an intensional arrow from A to B.

We will be particularly interested in cases where the comonad comes equipped with a way to regard every element of A as a "degenerate" computation in TA. We formalize this in the following definition.

Definition 2.3 A *computational comonad* over a category $\mathcal{C}$ is a quadruple $(T, \epsilon, \delta, \gamma)$ where (T, ϵ, δ) is a comonad over $\mathcal{C}$ and $\gamma : I_{\mathcal{C}} \dot{\to} T$ is a natural transformation such that, for every object A,

- $\epsilon_A \circ \gamma_A = \mathsf{id}_A$;
- $\gamma_{TA} \circ \gamma_A = \delta_A \circ \gamma_A$.

Naturality guarantees that, for every morphism $f : A \to^{\mathcal{C}} B$,

- $Tf \circ \gamma_A = \gamma_B \circ f$.

Figure 3 shows these properties in diagrammatic form. •

As an immediate corollary of these properties, ϵ_A is epi and γ_A is mono, for every object A. The existence of such a γ therefore subsumes (the dualized version of) Moggi's *computational monad* definition [Mog89], *i.e.*, that ϵ_A be epi.

Now, using γ, we can extract an extensional morphism from an intensional one, without lifting the type, as shown in the following proposition.

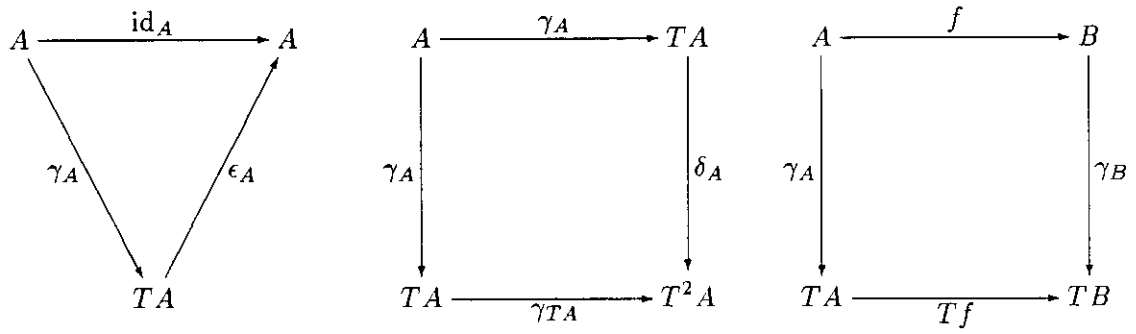

Figure 3: Properties of a computational comonad: these diagrams commute, for all A, B, and all morphisms $f : A \to^{\mathcal{C}} B$.

Proposition 2.4 *If $(T, \epsilon, \delta, \gamma)$ is a computational comonad, then there is a pair of functors (G, H) between $\mathcal{C}$ and $\mathcal{C}_T$ with the following definitions and properties:*

- *$G : \mathcal{C} \to \mathcal{C}_T$ is the identity on objects, and $Gf = f \circ \epsilon_A = \epsilon_B \circ Tf$ for every $f : A \to^{\mathcal{C}} B$.*
- *$H : \mathcal{C}_T \to \mathcal{C}$ is the identity on objects, and $Ha = a \circ \gamma_A = \epsilon_A \circ Fa \circ \gamma_A$, for all $a : A \to^{\mathcal{C}_T} B$.*
- *$H \circ G = I_{\mathcal{C}}$.*

The functoriality of H follows from the stated properties of γ.

We say that Ha is the *input-output morphism* of a. Note that for every f, the input-output morphism of Gf is f itself. In general, Gf is not the only intensional morphism whose input-output morphism is f (unless the intensional content of all morphisms is trivial, as would be the case if $TA = A$ for all A).

2.3 Cartesian Closure

Let $\mathcal{C}$ be a cartesian closed category with a distinguished product for each pair of objects A_1 and A_2, which we denote $A_1 \times A_2$. Clearly, a terminal object of $\mathcal{C}$ is also a terminal object of $\mathcal{C}_T$. Let (T, ϵ, δ) be a comonad over $\mathcal{C}$ that preserves finite products. Intuitively, this means that a pair of computations is (equivalent to) a computation of a pair. We omit the isomorphisms between $T(A_1 \times A_2)$ and $TA_1 \times TA_2$. It follows that distinguished product objects in $\mathcal{C}$ are also product objects in $\mathcal{C}_T$, Let $\pi_i : A_1 \times A_2 \to^{\mathcal{C}} A_i$ $(i = 1, 2)$ be projections in $\mathcal{C}$. Then the corresponding projections in $\mathcal{C}_T$ are given by:

$$\begin{aligned} \widehat{\pi}_i &: A_1 \times A_2 \to^{\mathcal{C}_T} A_i \\ \widehat{\pi}_i &= \epsilon_{A_i} \circ T\,\pi_i \\ &= \pi_i \circ \epsilon_{A_1 \times A_2} \\ &= G\,\pi_i\,. \end{aligned}$$

Pairing of morphisms in $\mathcal{C}_T$ is the pairing of morphisms in $\mathcal{C}$, and $T\,\langle f, g\rangle = \langle Tf, Tg\rangle$.

An exponentiation object $[TA \to^{\mathcal{C}} B]$ from $\mathcal{C}$ is also an exponentiation object $[A \to^{\mathcal{C}_T} B]$ in $\mathcal{C}_T$. The application morphism is given by:

$$\begin{aligned} \widehat{\mathsf{app}}_{A,B} &: [A \to^{\mathcal{C}_T} B] \times A \to^{\mathcal{C}_T} B \\ \widehat{\mathsf{app}}_{A,B} &= \mathsf{app}_{TA,B} \circ (\epsilon_{[TA \to^{\mathcal{C}} B]} \times \mathsf{id}_{TA}), \end{aligned}$$

where $\mathsf{app}_{TA,B} : [TA \to^{\mathcal{C}} B] \times TA \to^{\mathcal{C}} B$ is the corresponding application morphism in $\mathcal{C}$ and id_{TA} is the identity on TA in $\mathcal{C}$. Note that in general $\widehat{\mathsf{app}} \neq G(H\,\widehat{\mathsf{app}})$. We say more about this later. The currying of a morphism of $\mathcal{C}_T$ is its currying as a morphism of $\mathcal{C}$; this makes sense because $T(A \times B) \to^{\mathcal{C}} C$ is (isomorphic to) $TA \times TB \to^{\mathcal{C}} C$. The proof is straightforward: the desired universality property relating $\widehat{\mathsf{app}}$ and currying in $\mathcal{C}_T$ reduces to the same property of app and currying in $\mathcal{C}$, which holds by cartesian closure of $\mathcal{C}$:

$$\widehat{\mathsf{app}} \,\bar{\circ}(\mathsf{curry}(g) \times \widehat{\mathsf{id}}) \;=\; \mathsf{app} \circ (\mathsf{curry}(g) \times \mathsf{id}) \;=\; g.$$

Thus, we obtain the following result (see also [See89]):

Proposition 2.5 *If $\mathcal{C}$ is cartesian closed and the comonad T preserves finite products, then $\mathcal{C}_T$ is cartesian closed.*

3 Scott Domains, Paths and the Eagerness Ordering

We now fix as our extensional framework the category **Cont** of Scott domains [Sco82] (ω-algebraic, consistently complete, directed complete partial orders), with continuous functions as morphisms. It is well known that this category is cartesian closed. It is also a cpo-enriched category, since for each pair of domains D and D' the continuous functions from D to D' form a domain.

We define a notion of intensional behavior that involves an eagerness ordering between computations, where a computation over a domain D is a non-decreasing sequence of elements of D. To avoid a proliferation of parentheses, we use juxtaposition for function application (in **Cont**), and assume that function application has precedence over other operations. We use unadorned arrows for arrows in **Cont**.

Definition 3.1 A *path* over a domain D is an infinite non-decreasing sequence of elements of D, indexed starting from 1; we write s_i for the i-th element of s, for $i \geq 1$. Thus, for each $i \geq 1$ we have $s_i \leq_D s_{i+1}$.

We order paths over D componentwise: $s \preceq_D s'$ iff for every $i \geq 1$, $s_i \leq_D s'_i$. We call this the *eagerness ordering* on paths. •

Equivalently, paths are continuous functions from the positive integers domain (completed with a "point at infinity" and ordered by the usual "vertical" order) to D, and the eagerness ordering corresponds to the pointwise ordering of paths seen as such functions.

Note the following examples of paths over $\mathtt{Bool} \times \mathtt{Bool}$, where $\mathtt{Bool}$ is the flat domain of truth values:

$$\langle \bot, \bot \rangle^\omega \preceq \langle \bot, \bot \rangle \langle \bot, \bot \rangle \langle F, F \rangle^\omega \preceq \langle \bot, \bot \rangle \langle \bot, F \rangle \langle F, F \rangle^\omega \preceq \langle \bot, \bot \rangle \langle F, F \rangle^\omega \preceq \langle F, F \rangle^\omega ,$$

but the paths $\langle \bot, \bot \rangle \langle \bot, F \rangle \langle F, F \rangle^\omega$ and $\langle \bot, \bot \rangle \langle F, \bot \rangle \langle F, F \rangle^\omega$ are incomparable, even though they have the same lub.

For each D we define $\mathsf{val}_D\, s$ to be the $\leq_D$-lub of the elements of a path s, $\mathsf{val}_D\, s = \vee_D \{s_i \mid i \geq 1\}$. Intuitively, s represents a step-by-step computation to $\mathsf{val}_D\, s$.

Proposition 3.2 *The set PD of paths over a domain D, ordered by $\preceq_D$, forms a domain with least element the path $\bot^\omega_D$. A set of paths S is $\preceq_D$-consistent iff $\{\mathsf{val}\, s \mid s \in S\}$ is $\leq_D$-consistent, in which case S has a lub given by $\lambda i \,.\, \vee_D \{s_i \mid s \in S\}$. This also defines the lub of a directed set S. The finite elements of PD are the eventually constant paths built from finite elements of D.*

We complete P to a functor from **Cont** to **Cont** by letting Pf be $\mathsf{map}\, f$ for any $f : D \to D'$; $\mathsf{map} : (D \to D') \to PD \to PD'$ is defined by $\mathsf{map}\, fs = \lambda i \,.\, fs_i$. Clearly map and $\mathsf{map}\, f$ are continuous, for any continuous function f. Functoriality follows, because $\mathsf{map}\ \mathsf{id}_D = \mathsf{id}_{PD}$, and $\mathsf{map}(g \circ f) = \mathsf{map}\, g \circ \mathsf{map}\, f$ for all continuous functions f and g for which the composition makes sense.

For each D, we define pre_D, the *prefix computation* map, as follows:

$$\begin{array}{lcl} \mathsf{pre}_D & : & PD \to P^2 D \\ \mathsf{pre}_D & = & \lambda s \in PD \,.\, \mathsf{map}(\lambda x \in D \,.\, \mathsf{map}(\lambda y \in D \,.\, x \wedge_D y)s)s \\ & = & \lambda s \in PD \,.\, \lambda i \,.\, \lambda j \,.\, s_i \wedge_D s_j \\ & = & \lambda s \in PD \,.\, \lambda i \,.\, \lambda j \,.\, s_{\min(i,j)}. \end{array}$$

This definition makes sense because s is a non-decreasing sequence. As the notation suggests, $\mathsf{pre}_D\, s$ is the sequence of prefixes of s. Intuitively, $\mathsf{pre}_D\, s$ shows how the computation s progresses step-by-step: the i-th element of $\mathsf{pre}_D\, s$ is the computation $s_1 s_2 \ldots s_{i-1} s_i^\omega$. Note that $\mathsf{val}_D(\mathsf{pre}_D\, s) = s$, and that pre_D is continuous.

Lemma 3.3 *The triple* $(P, \mathsf{val}, \mathsf{pre})$ *is a comonad, and* P *preserves finite products.*

Proof: It is easy to verify that $\mathsf{val} : P \stackrel{\cdot}{\to} I_C$ and $\mathsf{pre} : P \stackrel{\cdot}{\to} P^2$ are natural transformations, and that they satisfy the associativity and identity constraints of a comonad. The functor P preserves finite products in **Cont** (up to an obvious isomorphism, which we can safely suppress), since there is an obvious way to synchronize a pair of paths – matching them componentwise – and obtain a path of pairs, and, conversely, every path of pairs determines a pair of paths by componentwise projection. A path in $P(D_1 \times D_2)$ is uniquely determined by its projections[1]. ∎

4 The Category Alg of Algorithms

Let **Alg** be the co-Kleisli category $\mathbf{Cont}_P$ for the comonad $(P, \mathsf{val}, \mathsf{pre})$ over **Cont**. We call the morphisms of **Alg** *algorithms*, and use the arrow $\to^i$ for algorithms. By the above development, since **Cont** is cartesian closed and P preserves products, it follows that **Alg** is cartesian closed. Spelling out in detail some of the implications:

- An algorithm from D to D' is a continuous function from PD to D'.
- The identity algorithm $\widehat{\mathsf{id}}_D$ from D to D is val_D.
- The composition $a' \mathbin{\bar{\circ}} a$ of two algorithms $a : D \to^i D'$ and $a' : D' \to^i D''$ is

  $$a' \mathbin{\bar{\circ}} a = a' \circ \mathsf{map}\, a \circ \mathsf{pre}_D \,.$$

  Composition is a continuous function on algorithms. Intuitively, given an input computation s over D, $(\mathsf{map}\, a \circ \mathsf{pre}_D)s$ is a computation over D' obtained by applying algorithm a successively to the partial results of the input computation, and this is a suitable argument for a'.
- **Alg** has the same distinguished terminal object as **Cont**.

[1] This would not be true, however, if paths were *strictly* increasing, since then the projections would lose synchronization information.

- **Alg** has the same distinguished product objects as **Cont**, with projection algorithms $\widehat{\pi}_i = \mathsf{val} \circ \mathsf{map}\,\pi_i = \pi_i \circ \mathsf{val}$, and the same pairing of morphisms.
- The exponentiation object $[D \to^i D']$ in **Alg** is the exponentiation $[PD \to D']$ in **Cont**. Algorithms form a domain under the pointwise ordering, which we denote $\leq^i$ and call the *intensional ordering* on algorithms, to emphasize the intensional setting.
- The application algorithm is given by:

$$\begin{array}{lcl} \widehat{\mathsf{app}} & : & (D \to^i D') \times D \to^i D' \\ \widehat{\mathsf{app}} & = & \mathsf{app} \circ (\mathsf{val}_{D \to^i D'} \times \mathsf{id}_{PD}) \;=\; \mathsf{app} \circ \langle \widehat{\pi}_1, \pi_2 \rangle\,. \end{array}$$

  Currying is the same as in **Cont**, and it is a continuous function on algorithms; the same is true for uncurrying.

4.1 Examples of Algorithms

Let `por`, `lor`, `ror` and `sor` be the parallel- , left-strict-, right-strict-, and doubly-strict-or functions from **Bool** $\times$ **Bool** to **Bool**. We introduce some algorithms for these functions.

Let `epPOR` be the least algorithm such that:

$$\begin{array}{lcl} \mathtt{epPOR}(\langle \bot, T \rangle^\omega) & = & T \\ \mathtt{epPOR}(\langle T, \bot \rangle^\omega) & = & T \\ \mathtt{epPOR}(\langle F, F \rangle^\omega) & = & F. \end{array}$$

Let `lpPOR` be the least algorithm such that (for all $n \geq 0$):

$$\begin{array}{lcl} \mathtt{lpPOR}(\langle \bot, \bot \rangle^n \langle \bot, T \rangle^\omega) & = & T \\ \mathtt{lpPOR}(\langle \bot, \bot \rangle^n \langle T, \bot \rangle^\omega) & = & T \\ \mathtt{lpPOR}(\langle \bot, \bot \rangle^n \langle F, F \rangle^\omega) & = & F. \end{array}$$

Intuitively, `epPOR` is the "most eager (parallel) algorithm" and `lpPOR` is the "laziest (parallel) algorithm" for the parallel-or function `por`: `epPOR` has the most eager computation strategy, insisting that the desired input evaluation is completed by the first computation step; and `lpPOR` has the most lazy (or least restrictive) computation strategy, in that it will allow arbitrarily many idle steps during the input computation. Note that by monotonicity it follows that, for all $n, m \geq 0$:

$$\begin{array}{lcl} \mathtt{lpPOR}(\langle \bot, \bot \rangle^n \langle \bot, F \rangle^m \langle F, F \rangle^\omega) & = & F \\ \mathtt{lpPOR}(\langle \bot, \bot \rangle^n \langle F, \bot \rangle^m \langle F, F \rangle^\omega) & = & F. \end{array}$$

In fact, `lpPOR` maps any computation producing a T in either input to T, and any computation producing F in both inputs to F; in contrast, `epPOR` maps a computation producing T in either input *on the first step* to T, and a computation producing F in both inputs *on the first step* to F; all other computations are mapped to $\bot$.

Note that `epPOR` $\leq^i$ `lpPOR`.

Let `esLOR` be the "most eager sequential algorithm" for the left-strict-or function, characterized as the least algorithm such that:

$$\begin{array}{lcl} \mathtt{esLOR}(\langle T, \bot \rangle^\omega) & = & T \\ \mathtt{esLOR}(\langle F, \bot \rangle \langle F, T \rangle^\omega) & = & T \\ \mathtt{esLOR}(\langle F, \bot \rangle \langle F, F \rangle^\omega) & = & F. \end{array}$$

Informally, we call this a *sequential* algorithm because each of its minimal output-producing computations makes incremental steps in a left-first way: in each of these computations the left input is evaluated first, and only if the left input is F is the right input evaluated. However, in this paper we will not give a more formal treatment of sequentiality. Again we call this algorithm eager because its computation strategy insists that the increments take place as early as possible (subject to the sequential order of evaluation).

Let epLOR be the "most eager parallel algorithm" for left-strict-or, the least algorithm such that:

$$\begin{aligned}
\texttt{epLOR}(\langle T, \bot\rangle^\omega) &= T \\
\texttt{epLOR}(\langle F, T\rangle^\omega) &= T \\
\texttt{epLOR}(\langle F, F\rangle^\omega) &= F.
\end{aligned}$$

Let lpLOR be the laziest parallel algorithm for the left-strict-or function lor, characterized as the least algorithm such that (for all $n, m \geq 0$):

$$\begin{aligned}
\texttt{lpLOR}(\langle \bot, \bot\rangle^n \langle T, \bot\rangle^\omega) &= T \\
\texttt{lpLOR}(\langle \bot, \bot\rangle^n \langle F, \bot\rangle^m \langle F, T\rangle^\omega) &= T \\
\texttt{lpLOR}(\langle \bot, \bot\rangle^n \langle F, \bot\rangle^m \langle F, F\rangle^\omega) &= F.
\end{aligned}$$

Note that epLOR $\leq^i$ esLOR $\leq^i$ lpLOR, and that epLOR $\leq^i$ epPOR and lpLOR $\leq^i$ lpPOR.

Algorithms epROR, esROR and lpROR for the right-strict-or function may be defined by analogy with the definitions given above for left-strict-or. We omit the details.

For the doubly-strict-or function sor we list four algorithms: epSOR, the most eager parallel; lpSOR, the laziest parallel; elrSOR, the most eager left-right sequential; and erlSOR, the most eager right-left sequential. The first two of these are defined in an analogous way to epLOR and lpLOR. The last two are characterized as the least algorithms satisfying the following conditions:

$$\begin{aligned}
\texttt{elrSOR}(\langle T, \bot\rangle \langle T, T\rangle^\omega) &= T \\
\texttt{elrSOR}(\langle T, \bot\rangle \langle T, F\rangle^\omega) &= T \\
\texttt{elrSOR}(\langle F, \bot\rangle \langle F, T\rangle^\omega) &= T \\
\texttt{elrSOR}(\langle F, \bot\rangle \langle F, F\rangle^\omega) &= F \\
\texttt{erlSOR}(\langle \bot, T\rangle \langle T, T\rangle^\omega) &= T \\
\texttt{erlSOR}(\langle \bot, T\rangle \langle F, T\rangle^\omega) &= T \\
\texttt{erlSOR}(\langle \bot, F\rangle \langle T, F\rangle^\omega) &= T \\
\texttt{erlSOR}(\langle \bot, F\rangle \langle F, F\rangle^\omega) &= F.
\end{aligned}$$

Note that epSOR $\leq^i$ elrSOR $\leq^i$ lpSOR and that epSOR $\leq^i$ erlSOR $\leq^i$ lpSOR, but the algorithms elrSOR and erlSOR are incomparable (since they have irreconcilable computation strategies).

Figure 4 summarizes the intensional ordering relationships between these or-algorithms [2]

4.2 Examples of composition

The composition of esLOR : $\mathbf{Bool}^2 \to \mathbf{Bool}$ and curry(lpPOR) : $\mathbf{Bool} \to (\mathbf{Bool} \to \mathbf{Bool})$ produces an algorithm $a : \mathbf{Bool}^2 \to \mathbf{Bool} \to \mathbf{Bool}$ such that, for all $n \geq 0$:

$$\begin{aligned}
a(\langle \bot, \bot\rangle^\omega)(\bot^n T^\omega) &= T \\
a(\langle T, \bot\rangle^\omega)(\bot^\omega) &= T \\
a(\langle F, \bot\rangle \langle F, T\rangle^\omega)(\bot^\omega) &= T \\
a(\langle F, \bot\rangle \langle F, F\rangle^\omega)(\bot^n F^\omega) &= F.
\end{aligned}$$

[2] Of course, these algorithms do not constitute a complete classification of all algorithms for the or-functions. We chose what we regard as a representative sample.

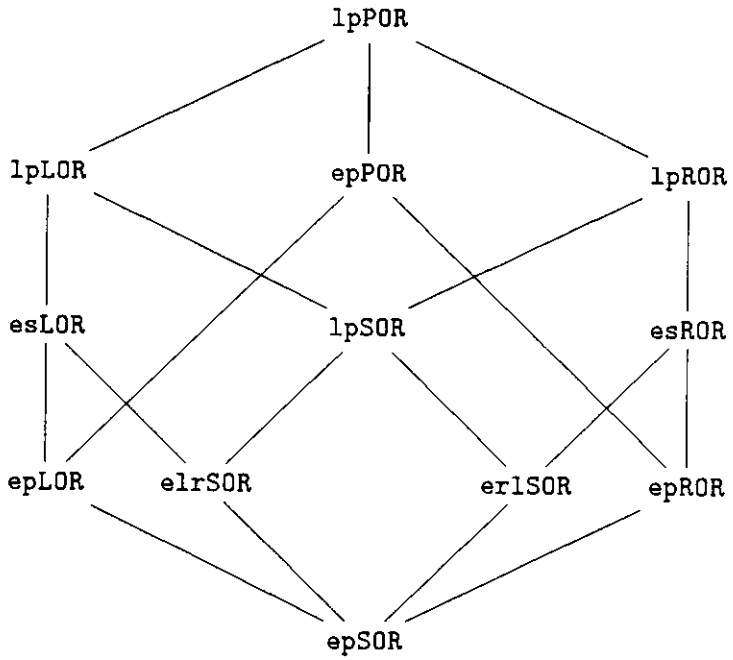

Figure 4: Some algorithms for or-functions, ordered intensionally

Let lNEG and eNEG be the least algorithms from **Bool** to **Bool** such that, for all $n \geq 0$,

$$\begin{aligned} \texttt{lNEG}(\bot^n T^\omega) &= F \\ \texttt{lNEG}(\bot^n F^\omega) &= T \\ \texttt{eNEG}(T^\omega) &= F \\ \texttt{eNEG}(F^\omega) &= T. \end{aligned}$$

The second-order function $\lambda a : \mathbf{Bool}^2 \to \mathbf{Bool}$. $\texttt{lNEG} \bar{\circ} a \bar{\circ} (\texttt{lNEG} \times \texttt{lNEG})$ transforms an algorithm a for a binary truth function into an algorithm for the dual of that function, and interchanges the roles of T and F in the algorithm's computation strategy. Let DUAL be the canonical algorithm for this function. The result of applying this algorithm to esLOR is esLAND, the most eager sequential algorithm for left-strict-and. Note that $\texttt{lNEG} \,\bar{\circ}\, \texttt{lNEG} = \widehat{\mathsf{id}}$, so that $\texttt{DUAL} \,\bar{\circ}\, \texttt{DUAL} = \widehat{\mathsf{id}}$ also.

If we replace lNEG by eNEG in the dualizer the effect would be the same on the function part of a but the computation strategy would be more radically altered (possibly becoming more eager). For example, with the obvious notation,

$$\texttt{eNEG} \,\bar{\circ}\, \texttt{lpPOR} \,\bar{\circ}\, (\texttt{eNEG} \times \texttt{eNEG}) \;=\; \texttt{epAND}.$$

5 Relating Algorithms and Continuous Functions

5.1 Relating Alg and Cont

There is an obvious way to regard a value in D as a degenerate computation over D, by identifying the value $x \in D$ with the constant path $x^\omega \in PD$. This is in fact the maximal path (in the eagerness ordering on paths) with lub x. We use this idea to turn $(P, \mathsf{val}, \mathsf{pre})$ into a computational comonad.

For each D let $\mathsf{path}_D : D \to PD$ be defined by:

$$\mathsf{path}_D = \lambda x \in D \,.\, \lambda i \,.\, x.$$

Simply, $\mathsf{path}_D\, x$ is the constant path to x. It is easy to check that this defines a natural transformation $\mathsf{path} : I_{\mathbf{Cont}} \stackrel{\cdot}{\to} P$. Obviously, for all D the function path_D is continuous, and we have the identities:

- $\mathsf{val}_D \circ \mathsf{path}_D = \mathsf{id}_D$,
- $\mathsf{path}_{PD} \circ \mathsf{path}_D = \mathsf{pre}_D \circ \mathsf{path}_D$.

By a special case of naturality, whenever $a : PD \to D'$ we have:

- $Pa \circ \mathsf{path}_{PD} = \mathsf{path}_{D'} \circ a$.

Thus $(P, \mathsf{val}, \mathsf{pre}, \mathsf{path})$ is a computational comonad (Definition 2.3).

Hence, by Proposition 2.4, there exist a canonical functor $G : \mathbf{Cont} \to \mathbf{Alg}$ and an input-output functor $H : \mathbf{Alg} \to \mathbf{Cont}$. They are both identity on objects, so that we restrict our attention to their morphism parts, which we call alg and fun, respectively:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathsf{fun} & : & (D \to^i D') \to (D \to D') & \mathsf{alg} & : & (D \to D') \to (D \to^i D') \\ \mathsf{fun}\, a & = & a \circ \mathsf{path}_D & \mathsf{alg}\, f & = & f \circ \mathsf{val}_D \end{array}$$

By functoriality, we have the following identities:

- $\mathsf{fun}\, \widehat{\mathsf{id}}_D = \mathsf{id}_D$
- $\mathsf{fun}(a' \mathbin{\bar{\circ}} a) = \mathsf{fun}(a') \circ \mathsf{fun}(a)$
- $\mathsf{alg}\, \mathsf{id}_D = \widehat{\mathsf{id}}_D$
- $\mathsf{alg}(f' \circ f) = \mathsf{alg}(f') \mathbin{\bar{\circ}} \mathsf{alg}(f)$

We say that $\mathsf{fun}\, a$ is the *input-output function* of a, or that a is an algorithm for $\mathsf{fun}\, a$. Intuitively, $\mathsf{fun}\, a$ is obtained by ignoring the internal path structure of a's arguments, by applying a only to constant paths.

We say that $\mathsf{alg}\, f$ is the canonical algorithm for f. Intuitively, $\mathsf{alg}\, f$ is oblivious to all but the final result of a computation; any computation to an input x will produce fx as output. Other algorithms for the same function may be more stringent, and output fx only for a subset of the computations to x.

Proposition 5.1 *For all D and D', all $f \in D \to D'$, and all $a \in D \to^i D'$,*

$$\mathsf{fun}(\mathsf{alg}\, f) = f,$$
$$a \leq^i \mathsf{alg}(\mathsf{fun}\, a).$$

Thus, $D \to D'$ is a retract of $D \to^i D'$.

In fact, since we also have the inequality $\mathsf{id}_{PD} \leq \mathsf{path}_D \circ \mathsf{val}_D$, each value domain D is a retract of the corresponding path domain PD.

Definition 5.2 We say that a_1 is *input-output below* a_2, written $a_1 \leq^{io} a_2$, iff $\mathsf{fun}\, a_1 \leq \mathsf{fun}\, a_2$.

Two algorithms a_1 and a_2 are *input-output equivalent*, written $a_1 =^{io} a_2$, iff they have the same input-output function. •

Proposition 5.3 *For all a_1, a_2, $a_1 \leq^i a_2$ implies $a_1 \leq^{io} a_2$.*

Proof: By monotonicity of fun. ∎

Note that the converse fails: $a_1 \leq^{io} a_2$ does not always imply $a_1 \leq^i a_2$. This is shown by any pair of distinct algorithms for the same function (such as `elrSOR` and `erlSOR`). This result indicates that the intensional order on algorithms takes into account intensional aspects of algorithms, whereas the input-output order does not.

Proposition 5.4 *For all D and D', the quotient of $(D \to^i D', \leq^i)$ by input-output equivalence is order isomorphic to $(D \to D', \leq)$, with* fun *and* alg *inducing the isomorphisms:*

$$(D \to^i D', \leq^i)/_{=^{io}} \cong (D \to D', \leq).$$

We say that an algorithm a is *canonical* iff $a = \mathsf{alg}(\mathsf{fun}\, a)$[3]. Since alg distributes over composition, the composition of two canonical algorithms is canonical.

Proposition 5.5 *The set of all algorithms for a given continuous function f forms a complete lattice under the pointwise ordering, with* alg f *as the top element:*

$$\mathsf{alg}\, f = f \circ \mathsf{val}_D = \vee^i\{a \mid \mathsf{fun}\, a = f\}.$$

We have input-output expressivity, in the following sense:

Corollary 5.6 *Every continuous function is the input-output function of some (canonical) algorithm.*

Maximality of alg f makes precise the sense in which we mean that it is the least stringent of all algorithms for f; in this setting, canonical may be read as "most lazy".

To illustrate these definitions and results, we return to the or-algorithms introduced earlier (Section 4.1). In each case the algorithm has the input-output function suggested by its name. For example, fun `epPOR` = fun `lpPOR` = por, and alg por = `lpPOR`. Moreover, `epPOR` is the least algorithm for the parallel-or function; leastness corresponds to our informal description of this as the most eager algorithm for this function. Similarly, fun `epSOR` = fun `elrSOR` = fun `erlSOR` = fun `lpSOR` = sor. Thus, these four algorithms are input-output equivalent.

Figure 5 shows the input-output equivalence classes of Figure 4, ordered by the quotient ordering $\leq^i /_{=^{io}}$. Within each equivalence class the figure also records the intensional ordering, to facilitate comparison with Figure 4. Note that if we identify each equivalence class with its (common) input-output function we obtain the pointwise order on these functions (Figure 6).

Recall that practically all operations we have defined are continuous, so that, by the above expressivity result, there are algorithms for each of these operations. For instance, there is a canonical algorithm $\hat{\circ}$ for algorithm composition. Some of the algorithms that were used to define the category **Alg** are canonical – $\widehat{\mathsf{id}}_D = \mathsf{alg}\,\mathsf{id}_D$, $\hat{\pi}_i = \mathsf{alg}\,\pi_i$. The important exception is the

[3] Equivalently, a is canonical iff there is a continuous function f such that $a = \mathsf{alg}(f)$.

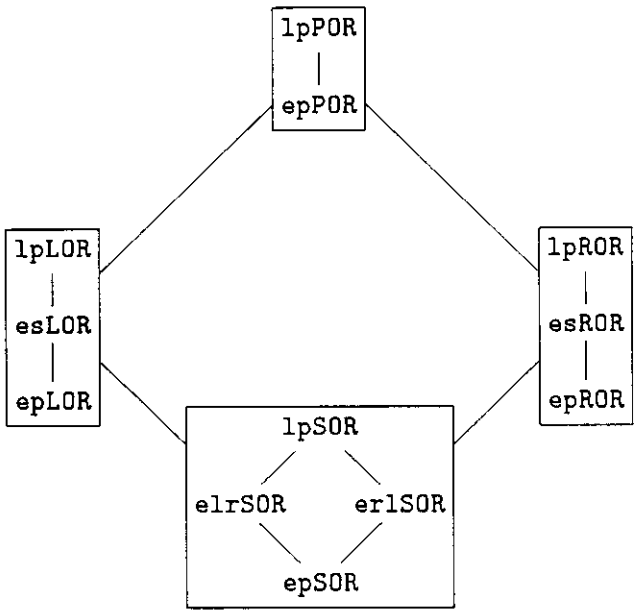

Figure 5: The or-algorithms, quotiented by input-output equivalence

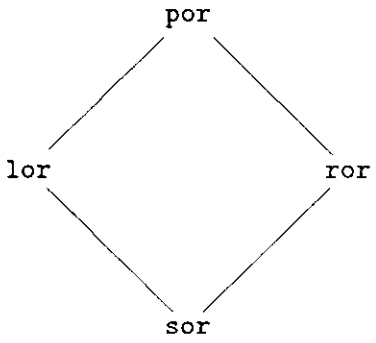

Figure 6: The or-functions, ordered pointwise

application algorithm $\widehat{\mathsf{app}}$, which is not canonical. Let $\overline{\mathsf{app}}$ be the input-output function of $\widehat{\mathsf{app}}$; then we have:

$$\begin{aligned}
\widehat{\mathsf{app}} &= \lambda(s,t)\ .\ \mathsf{val}\,st \\
&= \mathsf{uncurry}(\lambda s\ .\ \lambda t\ .\ \mathsf{val}\,st) \\
&= \mathsf{uncurry}\ \mathsf{val} = \mathsf{uncurry}\ \widehat{\mathsf{id}} \\
\overline{\mathsf{app}} &= \mathsf{fun}\,\widehat{\mathsf{app}} \\
&= \lambda(a,x)\ .\ a(\mathsf{path}\,x) \\
&= \mathsf{uncurry}(\lambda a\ .\ \lambda x\ .\ a(\mathsf{path}\,x)) \\
&= \mathsf{uncurry}(\mathsf{fun}) \\
\mathsf{alg}\,\overline{\mathsf{app}} &= \lambda(s,t)\ .\ (\mathsf{val}\,s)(\mathsf{path} \circ \mathsf{val}\,t).
\end{aligned}$$

In words, $\widehat{\mathsf{app}}$ is uniquely determined, as the uncurrying of the identity algorithm. But $\mathsf{alg}\,\overline{\mathsf{app}}$ ignores the computation of its second argument, and $\mathsf{alg}\,\overline{\mathsf{app}} \neq \widehat{\mathsf{app}}$; intuitively, this reflects the fact that $\overline{\mathsf{app}}$ only determines the *input-output* aspect of algorithm application, and, in fact, $\overline{\mathsf{app}}$ is the uncurrying of fun.

5.2 Intensional and Extensional Models

We define a simple type system by:

$$\tau ::= \rho \mid \tau_1 \times \tau_2 \mid \tau \to \tau',$$

where $\rho \in \mathbf{Atomic}$ ranges over a set of *atomic types* and τ ranges over the set **Type** of types. We define an extensional model E and an intensional model I for this type system in **Cont** and **Alg** respectively, each model being a type-indexed family of Scott domains. We assume given an interpretation $A\rho$ for each atomic type ρ, common to both the extensional and the intensional models.

Definition 5.7 The extensional model E, based on the category **Cont**, is the family of domains used in the standard continuous functions semantics; the intensional model I is its obvious analogue in **Alg**:

$$\begin{aligned}
E\rho &= A\rho & I\rho &= A\rho \\
E(\tau_1 \times \tau_2) &= E\tau_1 \times E\tau_2 & I(\tau_1 \times \tau_2) &= I\tau_1 \times I\tau_2 \\
E(\tau \to \tau') &= [E\tau \to E\tau'] & I(\tau \to \tau') &= [I\tau \to^i I\tau']
\end{aligned}$$

Of course the products and exponentiations here are taken in the appropriate category. •

With these definitions in hand, note that we have already seen that fun and alg can be used to relate $I(\tau \to \tau')$ and $[I\tau \to I\tau']$. We would like to go further, and relate $I(\tau \to \tau')$ and $E(\tau \to \tau')$. We achieve this as follows.

Definition 5.8 Define two type-indexed families of functions

$$\mathsf{ext}_\tau : I\tau \to E\tau \qquad \mathsf{int}_\tau : E\tau \to I\tau$$

by induction on τ:

- For $\rho \in \mathbf{Atomic}$, $E\rho = I\rho = A\rho$, and we let both ext_ρ and int_ρ be the identity function.

- For product types we define:

$$\mathsf{ext}_{\tau_1 \times \tau_2} = \mathsf{ext}_{\tau_1} \times \mathsf{ext}_{\tau_2} \qquad \mathsf{int}_{\tau_1 \times \tau_2} = \mathsf{int}_{\tau_1} \times \mathsf{int}_{\tau_2} .$$

- For an exponentiation $\tau \to \tau'$, we make use of fun and alg:

$$\begin{array}{rcl} \mathsf{ext}_{\tau \to \tau'} & = & \lambda a \ . \ \mathsf{ext}_{\tau'} \circ \mathsf{fun}\, a \circ \mathsf{int}_{\tau} \\ \mathsf{int}_{\tau \to \tau'} & = & \lambda f \ . \ \mathsf{alg}(\mathsf{int}_{\tau'} \circ f \circ \mathsf{ext}_{\tau}). \end{array}$$

•

For each $a \in I\tau$, we refer to $\mathsf{ext}_\tau(a)$ as the *extension* of a. Similarly, for each $e \in E\tau$, $\mathsf{int}_\tau(e)$ is the *intension* of e. Elements of atomic types have no (extra) intensional content.

Proposition 5.9 *For each τ, all $e \in E\tau$ and all $a \in I\tau$, we have:*

$$e = \mathsf{ext}_\tau(\mathsf{int}_\tau\, e),$$
$$a \leq^i \mathsf{int}_\tau(\mathsf{ext}_\tau\, a).$$

Thus the extension of $\mathsf{int}_\tau\, e$ is e, and every extensional value is the extension of some intensional value. Moreover, for each τ, $E\tau$ is a retract of $I\tau$.

We say that an element $a \in I\tau$ is *extensional* iff $a = \mathsf{int}_\tau(\mathsf{ext}_\tau\, a)$. Obviously, an extensional element is uniquely determined by its extension. Moreover, an extensional element is maximal among all elements with a given extension. An extensional algorithm is also canonical. It follows that the application algorithm $\widehat{\mathsf{app}}$ is neither extensional nor canonical.

Definition 5.10 When $a_1, a_2 \in I(\tau)$, we say that a_1 is *extensionally below* a_2, written $a_1 \leq^e a_2$, iff $\mathsf{ext}_\tau\, a_1 \leq \mathsf{ext}_\tau\, a_2$. Similarly, a_1 and a_2 are *extensionally equivalent*, written $a_1 =^e a_2$, iff they have the same extension. •

Proposition 5.11 *For all τ, and all $a_1, a_2 \in I(\tau)$, $a_1 \leq^i a_2$ implies $a_1 \leq^e a_2$. Hence, the quotient of $I(\tau)$ by extensional equivalence is isomorphic to $E(\tau)$, with ext_τ and int_τ inducing the isomorphism. That is,*

$$(I(\tau), \leq^i)/_{=^e} \ \cong \ (E(\tau), \leq).$$

Proposition 5.12 *For all τ, τ' and all $a_1, a_2 \in I(\tau \to \tau')$, $a_1 \leq^{io} a_2$ implies $a_1 \leq^e a_2$. Hence, an extensional equivalence class is a union of input-output equivalence classes.*

Let $\mathbf{Cont}_E$ and $\mathbf{Alg}_I$ be the full subcategories of **Cont** and **Alg**, respectively, obtained by taking only the domains that interpret some type. They are both cartesian closed, inheriting the exponentiation from their "parent" categories. It is interesting to note, however, that if we take the subcategory of $\mathbf{Alg}_I$ consisting of all objects but just the extensional algorithms, we get a cartesian closed category, but with a different structure – the structure of $\mathbf{Cont}_E$ – because each extensional algorithm is uniquely determined by its extension; the canonical application algorithm $\mathsf{alg}\,\overline{\mathsf{app}}$ satisfies the cartesian closure requirements in this subcategory although it does not in **Alg**.

5.3 Intensional and Extensional Semantics

We now consider a simply typed λ-calculus (with products) whose abstract syntax is summarized by:

$$M ::= c \mid X : \tau \mid M_1 M_2 \mid \lambda X : \tau . M \mid (M_1, M_2) \mid \text{fst } M \mid \text{snd } M$$

where c ranges over a set **Con** of (typed) constants and X ranges over a set **Ide** of identifiers. Let **Term** be the set of well-typed expressions conforming to this syntax, with the usual typing rules.

It is well known that one can define a straightforward semantics for the simply typed λ-calculus in a cartesian closed category, starting with interpretations for the constants and then interpreting abstraction and application by means of the categorical structure in the obvious way. We thus define a (standard) intensional semantics $\mathcal{I}$ using I and a (standard) extensional semantics $\mathcal{E}$ using E.

Let $U_I = \bigcup_{\tau \in \mathbf{Type}} I\tau$ and $U_E = \bigcup_{\tau \in \mathbf{Type}} E\tau$ be the intensional and extensional universes. An *intensional environment* maps identifiers into the intensional universe, and an *extensional environment* maps identifiers into the extensional universe:

$$\mathbf{Env}_E = \mathbf{Ide} \to U_E \qquad \mathbf{Env}_E = \mathbf{Ide} \to U_I.$$

We use ϵ to range over $\mathbf{Env}_E$ and ι to range over $\mathbf{Env}_I$. We say that ϵ (respectively, ι) respects types (for M) iff, for all $X : \tau$ occurring free in M, $\epsilon[\![X]\!] \in E\tau$ (respectively, $\iota[\![X]\!] \in I\tau$).

We assume given an extensional semantic function $\mathcal{A}_E : \mathbf{Con} \to U_E$ and an intensional semantic function $\mathcal{A}_I : \mathbf{Con} \to U_I$.

Definition 5.13 The extensional semantic function $\mathcal{E} : \mathbf{Term} \to \mathbf{Env}_E \to U_E$ is given by:

$$\begin{aligned}
&\mathcal{E}[\![c]\!]\epsilon = \mathcal{A}_E[\![c]\!] \\
&\mathcal{E}[\![X : \tau]\!]\epsilon = \epsilon[\![X]\!] \\
&\mathcal{E}[\![M_1 M_2]\!]\epsilon = (\mathcal{E}[\![M_1]\!]\epsilon)(\mathcal{E}[\![M_2]\!]\epsilon) \\
&\mathcal{E}[\![\lambda X : \tau . M]\!]\epsilon = \lambda e \in E\tau . \mathcal{E}[\![M]\!](\epsilon[e/X]) \\
&\mathcal{E}[\![(M_1, M_2)]\!]\epsilon = (\mathcal{E}[\![M_1]\!]\epsilon, \mathcal{E}[\![M_2]\!]\epsilon) \\
&\mathcal{E}[\![\text{fst } M]\!]\epsilon = \pi_1(\mathcal{E}[\![M]\!]\epsilon) \\
&\mathcal{E}[\![\text{snd } M]\!]\epsilon = \pi_2(\mathcal{E}[\![M]\!]\epsilon)
\end{aligned}$$

•

It is standard that, for all $M : \tau$ and all ϵ respecting types, $\mathcal{E}[\![M]\!]\epsilon \in E\tau$.

Definition 5.14 The intensional semantic function $\mathcal{I} : \mathbf{Term} \to \mathbf{Env}_I \to U_I$ is given by:

$$\begin{aligned}
&\mathcal{I}[\![c]\!]\iota = \mathcal{A}_I[\![c]\!] \\
&\mathcal{I}[\![X : \tau]\!]\iota = \iota[\![X]\!] \\
&\mathcal{I}[\![M_1 M_2]\!]\iota = (\mathcal{I}[\![M_1]\!]\iota)(\mathsf{path}(\mathcal{I}[\![M_2]\!]\iota)) \\
&\mathcal{I}[\![\lambda X : \tau . M]\!]\iota = \lambda p \in PI\tau . \mathcal{I}[\![M]\!](\iota[\mathsf{val}\, p/X]) \\
&\mathcal{I}[\![(M_1, M_2)]\!]\iota = (\mathcal{I}[\![M_1]\!]\iota, \mathcal{I}[\![M_2]\!]\iota) \\
&\mathcal{I}[\![\text{fst } M]\!]\iota = \widehat{\pi}_1(\mathsf{path}\, \mathcal{I}[\![M]\!]\iota) = \pi_1(\mathcal{I}[\![M]\!]\iota) \\
&\mathcal{I}[\![\text{snd } M]\!]\iota = \widehat{\pi}_2(\mathsf{path}\, \mathcal{I}[\![M]\!]\iota) = \pi_2(\mathcal{I}[\![M]\!]\iota)
\end{aligned}$$

•

Note that the definition for $M_1 M_2$ uses application in the category **Alg**, since $\mathcal{I}[\![M_1]\!]\iota \in I(\tau \to \tau')$ is a function from paths over $I\tau$ to values in $I\tau'$, and **path** transforms the value $\mathcal{I}[\![M_2]\!]\iota \in I\tau$ into a path of the appropriate type.

Again, it is standard that for all $M : \tau$, and all ι respecting types, $\mathcal{I}[\![M]\!]\iota \in I\tau$.

5.4 Relating the two semantics

We have already seen that each $E\tau$ is a retract of the corresponding $I\tau$. Whenever M is a well-typed term and ϵ and ι respect types, we have $\mathcal{E}[\![M]\!]\epsilon \in E\tau$ and $\mathcal{I}[\![M]\!]\iota \in I\tau$. We would like to establish that the extensional value of M is just the extension of the intensional value of M, under reasonable assumptions. We do this as follows.

Definition 5.15 Define a type-indexed family of relations $\sim_\tau \subseteq I\tau \times E\tau$ by:

$$\begin{array}{rcl} \sim_\rho & = & \mathsf{id}_{A\rho} \\ \sim_{\tau_1 \times \tau_2} & = & \{((a_1,a_2),(e_1,e_2)) \mid a_1 \sim_{\tau_1} e_1 \;\&\; a_2 \sim_{\tau_2} e_2\} \\ \sim_{\tau \to \tau'} & = & \{(a,f) \mid \forall (i,e) \in I\tau \times E\tau.\; i \sim_\tau e \;\Rightarrow\; a(\mathsf{path}\, i) \sim_{\tau'} f(e)\} \end{array}$$

•

Intuitively, $\sim$ is the identity relation at atomic types, is defined componentwise at product types, and at arrow types is defined in the natural "logical" way, so that an algorithm a relates to a function f iff whenever i relates to e, then the result of applying a to i in **Alg** relates to the result of applying f to e. In fact, this family of relations constitutes a *logical relation* [Sta85] between our two models.

Proposition 5.16 *Algorithm compositions relate to function compositions, in that for all* $a \in I(\tau \to \tau'), a' \in I(\tau' \to \tau'')$ *and* $f \in E(\tau \to \tau'), f' \in E(\tau' \to \tau'')$,

$$a \sim_{\tau \to \tau'} f \;\&\; a' \sim_{\tau' \to \tau''} f' \;\Rightarrow (a' \mathbin{\bar{\circ}} a) \sim_{\tau \to \tau''} (f' \circ f).$$

Proof: This follows because of the identity $\mathsf{fun}(a' \mathbin{\bar{\circ}} a) = \mathsf{fun}(a') \circ \mathsf{fun}(a)$. ∎

Proposition 5.17 *Currying of algorithms corresponds to currying of functions, in that for all* $a \in I(\tau_1 \times \tau_2 \to \tau')$ *and* $f \in E(\tau_1 \times \tau_2 \to \tau')$,

$$a \sim_{\tau_1 \times \tau_2 \to \tau'} f \;\Rightarrow\; \mathsf{curry}(a) \sim_{\tau_1 \to (\tau_2 \to \tau')} \mathsf{curry}(f).$$

The next result relates ext_τ and int_τ to $\sim_\tau$, and shows that each $\sim_\tau$ is a (partial) function.

Proposition 5.18 *For all* τ, *and all* $a \in I\tau$ *and* $e \in E\tau$,

- $a \sim_\tau e \;\Rightarrow\; e = \mathsf{ext}_\tau\, a$.
- $\mathsf{int}_\tau\, e \sim_\tau e$.

Proof: By structural induction on τ. ∎

The $\sim_\tau$ relations connect intensional values with extensional values. We lift them to environments as follows. Define $\iota \sim \epsilon$ iff for all identifiers $X : \tau$, $\iota[\![X]\!] \sim_\tau \epsilon[\![X]\!]$.

Suppose that the intensional and extensional interpretations of each constant are related, that is, for all $\tau \in$ **Type** and all constants c of type τ, $\mathcal{A}_I[\![c]\!] \sim_\tau \mathcal{A}_E[\![c]\!]$. Then the intensional and extensional interpretations of all well-typed terms of the lambda calculus are related:

Proposition 5.19 *For all* $M : \tau$, $\iota \sim \epsilon \;\Rightarrow\; \mathcal{I}[\![M]\!]\iota \sim_\tau \mathcal{E}[\![M]\!]\epsilon$.

Proof: This result is actually an instance of the Fundamental Theorem of Logical Relations [Sta85]. We give a direct proof by structural induction on M.

- For $M = c$ this holds by our assumption on $\mathcal{A}_I$ and $\mathcal{A}_E$.
- For $M = X$ this follows because of the assumption that $\iota \sim \epsilon$.
- For $M = M_1 M_2 : \tau$, suppose true for $M_1 : \tau' \to \tau$ and for $M_2 : \tau'$. Thus,

  $$\mathcal{I}[\![M_1]\!]\iota \sim_{\tau' \to \tau} \mathcal{E}[\![M_1]\!]\epsilon,$$
  $$\mathcal{I}[\![M_2]\!]\iota \sim_{\tau'} \mathcal{E}[\![M_2]\!]\epsilon.$$

  Then, by definition of $\sim_{\tau' \to \tau}$ it follows that

  $$\begin{aligned} \mathcal{I}[\![M_1 M_2]\!]\iota &= (\mathcal{I}[\![M_1]\!]\iota)(\mathsf{path}(\mathcal{I}[\![M_2]\!]\iota)) \\ &\sim_\tau (\mathcal{E}[\![M_1]\!]\epsilon)(\mathcal{E}[\![M_2]\!]\epsilon) \\ &= \mathcal{E}[\![M_1 M_2]\!]\epsilon \end{aligned}$$

  as required.
- For $M = \lambda X : \tau'.M'$, $M' : \tau''$, $\tau = \tau' \to \tau''$, suppose $\iota \sim \epsilon$. Then it follows that whenever $a \sim_{\tau'} e$, we also get $\iota[a/X] \sim \epsilon[e/X]$. We need to show that

  $$(\mathcal{I}[\![\lambda X : \tau'.M']\!]\iota)(\mathsf{path}\, a) \sim_{\tau''} (\mathcal{E}[\![\lambda X : \tau'.M']\!]\epsilon)(e),$$

  whenever $a \sim_{\tau'} e$. This follows easily, since the induction hypothesis for M' gives

  $$\mathcal{I}[\![M']\!](\iota[a/X]) \sim_{\tau''} \mathcal{E}[\![M']\!](\epsilon[e/X]),$$

  whenever $a \sim_{\tau'} e$, and we have the identity $a = \mathsf{val}\,\mathsf{path}\, a$ for all $a \in I\tau'$.
- For (M_1, M_2), fst M, and snd M we omit the details, which are straightforward.

∎

As a corollary, for all $M : \tau$, the extension of $\mathcal{I}[\![M]\!]\iota$ is exactly $\mathcal{E}[\![M]\!]\epsilon$, provided we make the appropriate assumptions.

Corollary 5.20 *If for all* $c \in \mathbf{Con}$, $\mathcal{A}_I[\![c]\!] \sim \mathcal{A}_E[\![c]\!]$, *then for all* $\iota, \epsilon, M : \tau$,

$$\iota \sim \epsilon \;\Rightarrow\; \mathsf{ext}_\tau(\mathcal{I}[\![M]\!]\iota) = \mathcal{E}[\![M]\!]\epsilon.$$

This is the desired correspondence between intensional meanings and extensional meanings. Note that our semantic definitions and results are parameterized by the choice of the set **Con** of constants and their semantics. In particular, since the least fixed point operator is itself a continuous function, it is certainly possible to include constants Y_τ of type $(\tau \to \tau) \to \tau$ in order to handle recursive terms. The results still hold. We regard the correspondence established above as showing that the (standard) intensional semantics is conservative over the extensional semantics; if we regard the ext operator as removing the extra intensional information from its argument, the result states that the extensional semantics is faithfully embedded in the intensional one.

6 Future Work

We have examined in detail one particular notion of computation in the setting of Scott domains and continuous functions. It should be clear that our analysis can be adapted to other settings and other notions of intensional behavior where the necessary categorical and algebraic conditions hold.

Various restricted notions of continuous function have been used elsewhere, including Berry's stable functions [Ber78] and Kahn and Plotkin's sequential functions [KP78]. Various different kinds of semantic domains have been shown to be useful. We plan to investigate the possibility of emulating our algorithm construction when we vary the choice of underlying ccc, or the choice of ordering on paths, or even when we adopt a notion of computation farther removed from paths. It would be interesting to see to what extent some of the built-in assumptions could be relaxed, such as the property that the comonad preserves product. We are currently investigating what happens when we employ a comonad of strictly increasing paths; this leads to a model closely related to our earlier query model [BG90], but general enough to work at all types, not just first-order. Although cartesian closure fails, the category of algorithms still seems to provide a sensible intensional model for the λ-calculus.

The notion of stability makes sense as a restriction on the continuous functions between arbitrary domains, although it is mainly used when the underlying domains have certain extra properties (e.g. the dI-domains). It therefore seems natural to focus on the class of algorithms which are defined as stable functions from PD to D'. Here the notion of stability refers to the path ordering on arguments to an algorithm, and one might refer to such an algorithm as *intensionally stable*. The class of algorithms whose input-output function is stable is also very natural; we might call these the *extensionally stable* algorithms. We plan to investigate the properties of these classes of algorithms.

We would like to formulate rigorous notions of intensional and extensional sequentiality for algorithms. In order to do this we need to adopt a restricted notion of domain which supports a general definition of sequentiality. Concrete domains [KP78, Cur86] permit a suitable definition of sequentiality, but are not closed under exponentiation if morphisms are taken to be either continuous, or stable, or sequential functions. In a related paper [BG91] we introduce a generalization of the notion of concrete data structure that is closed under the continuous and stable function spaces, and which appears to support a sensible definition of sequentiality.

We would like to understand better the relationship between our development of algorithms and other work such as the computational λ-calculus of Moggi, the λ_p-calculus of Rosolini [Ros86], and the use of comonads in modelling the ! modality in linear logic [Laf88, Gir89]. Moggi states that his view of programs as "functions from values to computations" (leading to the use of monads and the Kleisli category) corresponds to call-by-value, and that an alternative view of programs as functions from computations to computations corresponds to call-by-name. We offer a third alternative: programs as functions from computations to values, leading to the use of comonads and the co-Kleisli category.

In this paper we have applied our ideas on intensional semantics to the simply typed λ-calculus. We worked out some of the properties of a "standard" intensional semantics, including the relationship with the standard extensional semantics. It should also be possible to use our intensional model to provide a non-standard interpretation for the λ-calculus, for instance by varying the choice of algorithm used to interpret application. It would be interesting to apply our ideas to languages with more explicitly intensional features, such as the CDS0 language of [Cur86].

7 Acknowledgements

Thanks to Manfred Droste and Yuri Gurevich for inviting us to present a preliminary version of this work at the Seminar on Programming Language Semantics and Model Theory, Dagstuhl Castle, June 23-29, 1991. Thanks also to John Reynolds, whose diagram macros made the drawing of our pictures much easier.

References

[Ber78] G. Berry. Stable models of typed λ-calculi. In *Proc. 5th Coll. on Automata, Languages and Programming. LNCS 62*, number 62 in LNCS, pages 72–89, Berlin, New-York, July 1978. Springer Verlag.

[BG90] S. Brookes and S. Geva. Towards a theory of parallel algorithms on concrete data structures. In *Semantics for Concurrency, Leicester 1990*, pages 116–136. Springer-Verlag, July 1990. Extended version to appear in Theoretical Computer Science.

[BG91] S. Brookes and S. Geva. Continuous functions and parallel algorithms on concrete data structures. To appear in Proceedings of MFPS91, Springer LNCS, 1991. Extended Abstract.

[Cur86] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. Pitman, London, 1986.

[Gir89] J.-Y. Girard. *Proofs and Types.* Cambridge University Press, 1989.

[KM77] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 1977*, pages 993–998. North Holland, 1977.

[KP78] G. Kahn and G. Plotkin. Domaines concrets. Rapport 336, IRIA-LABORIA, 1978.

[Laf88] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59, 1988.

[ML71] S. Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual IEEE Symposium on Logic in Computer Science (Asilomar, CA)*, pages 14–23. IEEE Computer Society Press, June 1989.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[Ros86] G Rosolini. Continuity and effectiveness in topoi. Ph. D. thesis, University of Oxford, 1986.

[Sco82] D. Scott. Domains for denotational semantics. In *Proceedings of ICALP 82*, volume 42 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, 1982.

[See89] R. A. G Seely. Linear logic, *-autonomous categories and cofree coalgebras. *Contemporary Mathematics*, 92:371–382, 1989.

[Sta85] R. Statman. Logical relations and the typed lambda calculus. *Information and Control*, 65:85–97, 1985.